LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-443

ON THE EFFICIENT EXPLOITATION OF SPECULATION UNDER DATAFLOW PARADIGMS OF CONTROL

Richard Mark Soley

May 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): MIT/LCS/TR-443	5. MONITORING ORGANIZATION REPORT NUMBER(S): N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION: MIT Laboratory for Computer Science; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code): 545 Technology Square, Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code): Information Systems Program, Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA/DOD; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Boulevard, Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification): On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control

12. PERSONAL AUTHOR(S): Soley, R.M.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1989 May	162

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	architecture, artificial intelligence, dataflow, I-structure storage, processes, search, speculation, tasks

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Dataflow architectures to exploit the parallelism in large scientific codes are now taking form. However, no approach to exploiting speculative, searching parallelism has been explored, even though (or perhaps because) the potential parallelism of such applications is tremendous. A view of speculation as a process which may proceed in parallel in a controlled fashion is explored, using examples from actual symbolic processing situations.

The central issue of exploiting this parallelism is the dynamic containment of the resources necessary to execute large speculative codes. We show efficient structures (graph schemata and architectural support) for executing highly speculative programs (such as expert systems) under a dataflow execution paradigm.

Controls over cross-procedure parallelism in an extensible manner will be presented, with applications to the various current problems of dataflow computation. Approaches to scheduling, prioritization and search tree pruning are considered, evaluated and compared.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code): (617) 253-5894; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1985—607-047

Unclassified

On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control

Richard Mark Soley

MIT / LCS / TR-443
May 19, 1989

On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control

Richard Mark Soley

Technical Report MIT / LCS / TR-443
May 19, 1989

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

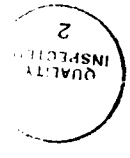
Abstract

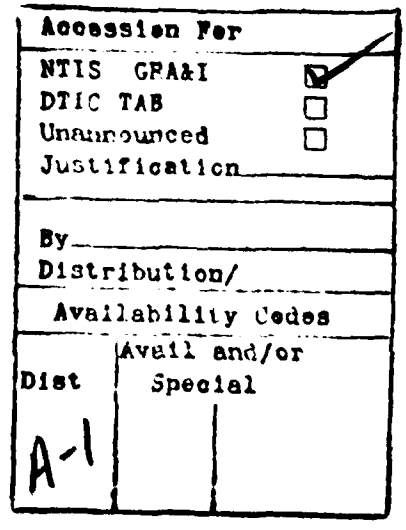
Dataflow architectures to exploit the parallelism in large scientific codes are now taking form. However, no approach to exploiting speculative, searching parallelism has been explored, even though (or perhaps because) the potential parallelism of such applications is tremendous. A view of speculation as a process which may proceed in parallel in a controlled fashion is explored, using examples from actual symbolic processing situations.

The central issue of exploiting this parallelism is the dynamic containment of the resources necessary to execute large speculative codes. We show efficient structures (graph schemata and architectural support) for executing highly speculative programs (such as expert systems) under a dataflow execution paradigm.

Controls over cross-procedure parallelism in an extensible manner will be presented, with applications to the various current problems of dataflow computation. Approaches to scheduling, prioritization and search tree pruning are considered, evaluated and compared.

Keywords: architecture, artificial intelligence, dataflow, I-structure storage, processes, search, speculation, tasks.





Acknowledgements

After eleven years at M.I.T., I've plenty of people to thank. These years (four undergraduate, seven as a graduate student) have been tremendously rewarding for me, and many friends have helped make it an interesting experience.

I would first like to thank Professor Arvind, my research and thesis supervisor and leader of the Computation Structures Group at the Laboratory for Computer Science. Besides his own considerable knowledge, his ideas have drawn many bright people to surround him in developing dataflow computer systems. His group, of which I have been a member for six and a half years, continues to be exciting and vital as it enters the "real dataflow machine" phase. I thank Arvind for his continued support of my research interests (and digressions) over the years.

Professors Robert H. ("Bert") Halstead, Jr. and Ramesh Patil, my readers, have been friends and guides for years. I thank them both for their patience in reading the thesis, and clear commentary on it.

Many friends and colleagues in the CSG research group contributed ideas, inspiration, support and proofreading to the completion of this thesis. In particular, I would like to thank Ken Traub, catalyst for the serialization ideas used in my I/O schemes and generally very bright guy;[†] Steve Heller, a slightly crazed person with whom I've thought about garbage collection and other assorted topics; Ken Steele, office mate extraordinaire and co-inventor of the lock protocol; Paul Barth, another lock co-inventor and scholar of programming with state; Paul Johnson, who read the thesis with a magnifying glass; Jonathan Young and Shail Aditya Gupta, my type tutors; and Jamey Hicks, Dana Henry and Michel Sadoune.

I also must thank others with whom I've worked closely on various projects over the years at CSG: Gregory Papadopoulos, who taught me the difference between a 74112 and a 74F112; David Culler, who pioneered resource management for dataflow machines; Dinarte R. Morais, primary author of Illustrate and the first Id World; the late Bhaskar GuhaRoy, the first person to think about I/O in Id; and Robert A. Iannucci, Rishiyur S. Nikhil, Andy Boughton, Paul Fuqua, and Poh Chuan Lim.

For nearly four years I was a part of the Mathlab group at LCS working on Macsyma and related systems; the people of the Mathlab Group gave me my start in computer systems research. I would particularly like to thank Carl Hoffman, Richard Zippel and Joel Moses for their support and interest. Particular thanks go to Joel for his delightful graduation gift.

[†] Despite his sesquipedalian characterization of my dissertation designation.

I would also like to thank my father Joseph Soley, and brothers David, Tim and Jack, who thought I'd never leave M.I.T. And also my aunt, uncle and cousins Judy, Bob, John and Jill Soley—who knew I would!

Alexander Michael Szabó Soley won't remember me working on this thesis, as he was born on October 4, 1988. But I will always remember these last few tough months and his sweet red-haired smile peering out of the playpen while I worked. Thank you, Alejo, for your smiling face!

> The child is the father of the man.
> —WILLIAM WORDSWORTH, *My Heart Leaps Up*

The most important person in my life is Isabel Thérèse Szabó Schwartz, my wife, friend and companion. This wonderful woman with whom I share my life, for now and always, is caring, sweet, loving, brilliant, beautiful, insightful and intuitive. Without her I certainly never would have finished this work, but that is the least of it. I thank her for her infinite patience in seeing me through my years at M.I.T. Isabel, I love you with all my heart.

> One word frees us of all the weight and pain of life:
> That word is love.
> —SOPHOCLES, *Œdipus at Colonus*

I dedicate this work to my wonderful wife and child (and future children), and in remembrance of my mother and mother-in-law.

This thesis was typeset in the Computer Modern Roman typeface by the TEX document preparation system,[48] using the LATEX macro package[52] as well as extensions written by the author and by Ken Traub. The illustrations and dedication were generated by the *Illustrate* document illustration system[72] using Computer Modern typefaces.

to

Barbara Lee Berman Soley

בְּרָכָה לְאָה ז״ל

and

Micheline Monique Veronique Schwarcz Szabó

מִרְיָם ז״ל

All that's bright must fade,–
The brightest still the fleetest;
All that's sweet was made
But to be lost when sweetest.

— *Thomas Moore*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Multiprocessing for Performance

Throughout the history of computing, researchers have attempted to increase computational resources focused on a single problem by usefully harnessing multiple central processing units to function as a single unit. Particularly when faced with the real or perceived physical limitations of the underlying physics of current computers, designers turn to the "obvious" solution of sharing a computational problem between two or more processing units.

Unfortunately, as parallel computer architects have discovered many times, most extant software schemas for expressing problem solutions do not transparently port to multiple processor environments. Indeed, not only the language but the entire approach to programming requires reconsideration in order to fully utilize most of the parallel processing equipment to date.

Two different paths have been followed to resolve this dilemma. Computer architects have concentrated most on finding holistic answers, perhaps including mild (instead of drastic) changes to the programming model, combined with a new system architecture. Most researchers in the area of Artificial Intelligence, however, have attempted to find new programming methodologies which allow direct expression (by the programmer) of the obvious parallelism in an algorithm.

The Artificial Intelligence (A. I.) approach to programming has traditionally been a hungry consumer of computational resources. Due to the explosive combinatorial nature of most

A. I. algorithms (such as in pattern matching, various search techniques, and large-scale knowledge representation), A. I. researchers are continually in search of a more powerful computational engine. The promise of increasing power by simply adding processing units is too much to ignore.

However, A. I. approaches rely on models of programming that differ in important details. In fact, the run-time resources necessary for both LISP and PROLOG are perceived by many A. I. programmers to preclude efficient implementation on system architectures designed for more standard computer languages. In addition, the programming styles affected by LISP programmers, and forced upon PROLOG programmers, are quite intensive (both in I/O and computation) in some areas not considered as important in the construction of conventional architectures. What is different about the A. I. style of programming that seems to draw so deeply on system resources? Architects of LISP and PROLOG systems see many answers to this question.

A strong argument is generally made for dependence on run-time typing (or tagging) of data values. In effect, the need to be able to determine the type of an object quickly during the execution of a LISP or PROLOG program may require type bits to be included in the object reference (*i.e.*, the pointer) itself. The additional design burden to avoid the overhead of extracting type information for frequent operations (such as arithmetic operations) accounts for most of the complexity of existing LISP implementations executing on conventional machine architectures.

In addition, both the LISP and PROLOG programming styles encourage programming with small procedures, relying on a fast function calling mechanism to allow the program to be broken up for ease of editing and debugging. This lends greatly to high programmer productivity in the LISP environment. LISP and PROLOG function calling mechanisms must also allow for dynamic name resolution[70, 26] ("dynamic linking") due to the generally incremental mode of program creation followed by A. I. adherants.

Garbage collection (the process of automatic reclamation of memory for re-use) also extends the demands of this class of programming languges. In order to support either LISP's or PROLOG's fully automatic dynamic storage allocation and reclamation facilities efficiently, hardware and operating system support are often necessary. This requirement may depend on standardized stack formats, tag bits in memory cells to disambiguate pointers from

other values, more involved memory allocation schemes, and even fast or parallel pointer arithmetic and comparison.

A. I. systems also are often I/O intensive; in fact, the tendency in expert system construction has been toward "do a little I/O, then do some computing" rather than the scientific computing approach of reading the dataset, computing, and ending execution. I/O is often central to A.I. systems.

An often-cited generalization of A. I. codes is that they deal primarily with the organization and reorganization of symbolic (*i.e.*, representational) relational data.[39] This is the meaning of the "symbol processing" moniker frequently pinned on the LISP language. Implementations of mappings from one representation to another (such as hash tables and the like), as well as categorization and directory systems, are common in knowledge representation systems.

Though many of these prerequisites were first found in A. I. programs, features of these languages have found their way into more "mainstream" languages and applications. Even BASIC interpreters, for example, require garbage collectors. The tendency to write small procedure blocks to enhance modularity and software reusability is spreading to many quarters, encouraging fast function call support. Heavy I/O demands are found in many non-A. I. application codes, such as air traffic control. Relational database schemes used in myriads of non-A. I. codes comprise many of the organizational and symbolic methods of knowledge representation. And pointer tagging has found its way into various machine architectures for protection and other purposes.

It is *speculation* that is usually overlooked in pointing out the differences in the A.I. approach to programming. Solving problems by searching spaces of knowledge, and thus "throwing away" computation in execution tracks that do not bear fruit, is a central requirement of most if not all A. I. systems. This is the important differentiating characterization of A. I. systems, indeed of search-based "heuristics" versus "algorithms." Even simple adaptive heuristics (such as virtual memory page replacement algoritms) may be said to be speculative in that they attempt to optimize resource usage for *future* events, based on *past* information that is not necessarily predictive.

It is important to note that speculative computation that doesn't contribute to the final result isn't really waste. Although the final result doesn't depend on it, what really counts

is the expected time to completion, not the amount of "useful" computation. Furthermore, the extra overhead of speculation is not necessarily great.

A simple example clarifies the point: a sequential search for an element of a n-length vector takes an expected $O(\frac{n}{2})$ computations to complete (worst case would be $O(n)$). The parallel "stupid" approach of using n processors, each testing the contents of one element of the vector, completes in one time step (i.e., $O(n)$ computations). The worst overhead of the parallel approach was a factor of two.

In addition, speculative approaches to programming are wonderful sources of parallelism.

1.2 Types of Parallelism

The parallelism inherent in any algorithm may be classified into two types; we call these *speculative parallelism* and *real parallelism*. An algorithm is said to contain real parallelism if two or more blocks of the algorithm, both of which are guaranteed to contribute to the final result, may proceed completely in parallel. This implies that there are no dependencies of any kind between the two blocks, but that their results are combined in some way to form the result of the overall computation. For example, in the evaluation of

$$(2 \times 3) + (4 \times 5)$$

the multiplications can clearly be done at the same time, as neither operation depends on the result of the other.

On the other hand, an algorithm is said to contain *speculative* parallelism if two or more blocks of the algorithm may proceed completely independently (as before), but *the results of one or more of these parallel "tracks" may be ignored by later portions of the algorithm* in forming the final result. A perfect example of speculative algorithm is searching an array for some element; clearly if one has as many processing elements as elements in the array, the array could be searched "in parallel" with each processor checking a different element of the array. However, assuming the element being searched for is found in only one position of the array, most of the computation performed was, in a sense, "needless." Thus the search algorithm is said to be speculative.

This situation is quite common in A. I. programming, due to the plethora of programming approaches using search or "generate and test" methodologies in problems implying large search spaces. The core of many A. I. programs contain nothing more than complicated search paradigms. In particular, rule-based expert systems generally consist of nothing more than a "recognize and act" structure, composed primarily of a pattern matching search of the rule base.

PROLOG's *backtracking*,[26] in fact, is just an implementation method for parallel speculation in a search tree. Attempted rule-matching could be said to be a parallel process in PROLOG, with backtracking implementing the scheduler for multiprogramming. PROLOG's cut operator is just an implementation control which depends on this sequentialization of the parallel processes to slow tree growth. This implied speculative control structure is PROLOG's great strength; programmers wishing to specify simply the logical structure of a problem rather than steps to its solution may do so. However, PROLOG's depth-first approach to solving the first-order logical problems posed is also its great weakness, as the approach constrains the user to a search paradigm that may not be correct for the domain. Indeed, Clocksin and Mellish concede[26] that PROLOG's strictly sequential semantics preclude it from meeting its goal of an automatic predicate calculus solving system.

Thus many A. I. systems builders have struck out in the direction of more parallel speculative structures.[25, 30, 3] Due to the clarity of the speculative parallelism, approaches to parallel computers designed by A.I. researchers have tended to emphasize the recognition of speculative parallelism to the exclusion of real parallelism. Control structures for meting out computational resources, and withdrawing such resources at such time as further speculation is unnecessary, then become the core of the parallel machine's structure.

Unfortunately, such an approach generally relies on specification of parallelism by the programmer. Worse, it leaves control of all system resources with the programmer, without offering tools for automating the process. Although this approach has been shown to function reasonably well, it fails miserably for large numbers of parallel processes or processors[33].

Dataflow control models of computation do not display this property. No explicit programmer specification of parallelism, and little or no specification of other system resources, are required under dynamic dataflow architectures such as the M.I.T. tagged-token class of dataflow machines[6, 61]. Recognition of fine-grained parallelism, at the operational level,

combined with functional languages such as ID [58], provide a basis for potentially scalable high-performance computers.

Most research into such computational models, however, has concentrated on large memory- and compute-intensive scientific programs[7]. Little view has been given to considering speculative styles of programming. In fact, because of the lack of the sequential ordering imposed by execution on serial computers, a straightforward implementation of some A. I. systems might result in *slower*, deadlocked or even *live-locked* machinery due to the overhead of managing a highly parallel computation.

The problem then becomes one of holding the explosive parallel growth of searches in A. I. systems to the bounds of the physical machine, and optionally prioritizing the order of those searches. In fact, some controls are necessary even in non-explosive contexts;[7, 28] similar controls must be found for the controlled serialization of speculation. This serialization requires a basic synchronization mechanism between the disjoint branches of the execution tree.

1.3 Synchronization Mechanisms

One of the unique features of the Tagged-Token Dataflow Architecture is that program synchronization, like the exploitation of program parallelism, is determined entirely by the compiler and run-time system, without the intervention of the programmer[6]. Although synchronization is a trivial problem in the general sequential-machine case (each instruction must wait for the termination of the linearly previous instruction, due to the implied full order of the instruction stream), the uncovering of potentially great amounts of parallelism in a program invokes a concomitant need to synchronize the far-flung portions of the execution tree (*i.e.*, to "join" forked execution branches). This need arises in any area that requires concurrent access to resources, including (1) explicit, static program joins; (2) dynamic memory access (I-structures[41]); (3) access to ordered *streams* of data[63]; (4) ordered access to input/output devices; and (5) control of execution graph growth itself.

Solutions for four of these synchronization problems under the tagged-token dataflow paradigm exist. These include the following:

- Concurrent program memory access is managed at compile time by explicitly joining execution tree branches (static links in the dataflow graph). The waiting-matching section of the general tagged-token architecture then completes the join operation at run time, regardless of input order.
- The I-structure controller portion of the tagged-token design[41, 14] implicitly handles all I-structure synchronization (dynamic links in the dataflow graph) at run time, maintaining atomicity of memory operations and completing dynamic links in the dataflow graph, again regardless of read/write order.
- Heller's work covers a solution for the correct execution of finite and infinite stream-oriented computation.[42]
- The author's paper on input/output in ID specifies a solution to the I/O ordering problem by using a combination of static and dynamic serialization of I/O-producing programs.[66]

Many controls over execution graph growth (in particular, loop resource usage) have appeared in Culler's work.[28] However, no complete management system for controlling execution graph growth has appeared for the tagged-token paradigm. In addition, no synchronization model for parallel, non-interdependent dataflow graphs (such as in speculative execution) has been developed. This report puts forward solutions for this problem.

1.4 The State of the Art

As noted previously, the idea of shortening the critical path of A. I. codes by taking advantage of speculative parallelism is not new. Here we review some of the state of the art in parallel A. I. systems, with special emphasis on several questions:

- Does the system take advantage of both real and speculative parallelism?
- Does the programmer need to specify the parallelism of the application, or does the compiler or machine recognize parallelism automatically?
- If speculative parallelism is allowed, is there a systematic manner in which parallel speculative work may be controlled? For example, if after a speculative "track" of computation is *known to be worthless* (*i.e.*, computing after an alternate answer has been found) may that computation be terminated?
- If control such as termination is allowed, are the semantics of the host language (the language in which the heuristic is specified) clarified in the event of termination?

Arguably the most popular family of languages in which A. I. applications are written is the LISP group of dialects, with the major offshoots SCHEME and COMMON LISP as the most

prominent exemplars.[1, 70] One of the most popular approaches to adding "parallelism specification" to this class of languages is the *FUTURE* declaration.[39, 54, 15]

One well-studied exemplar of the *FUTURE* approach is that of MULTILISP, a *FUTURE*-extended version of the SCHEME dialect.[39, 38] In MULTILISP, both real and speculative parallelism may be specified, and only through the *FUTURE* procedure. In effect, *FUTURE* in MULTILISP takes a single argument, and returns immediately (potentially before the argument has been evaluated) with a *future reference*, or *placeholder*, with which the calling program may access the value once it has been determined. For example,

```
(+ (future (* 2 3))
   (future (* 4 5)))
```

implements the real parallelism available in the simple mathematical expression we saw in Section 1.2. This explicitly-coded parallelism requires the interpreter to, potentially in parallel, determine the values of the expressions (2×3) and (4×5), and then sum those results. The *process* doing the addition will *spawn* the two multiplication processes, and immediately wait for those processes to finish. †

However, this is not wherein the real power of MULTILISP lies. The results returned from the *FUTURE* expressions of the program above are in some sense *"touched"* immediately; the addition process needs those values before it can compute. Consider instead the program

```
(cons 8 (future (expensive-computation)))
```

where **expensive-computation** may compute for a long time before it is complete. Unlike the addition in the previous program, the **cons** in this program may return *immediately* after constructing a LISP **cons** cell. The **car** of the cell will contain the requisite **8**, while the **cdr** of the cell will contain a *future*. In other words, since **cons** only *stores* its arguments, and doesn't require the "value" of those arguments, it may be "lazy" about the actual value of its arguments.‡ In addition to specifying parallelism, futures allow specification of

†Of course, there is some useless expression of parallelism in this program, since the addition process has nothing to do while the multiplications are going on.

‡This laziness is not the same as the laziness discussed by Henderson[43] or Heller[42], although the concepts are related.

argument non-strictness on an argument-by-argument basis (*i.e.*, procedures may be defined such as to return valid results with invalid or error-value inputs, or even before such inputs are available).

The *FUTURE* construct allows us to explicitly specify as much real or speculative parallelism as we wish. Programs using MULTILISP for every function, including sorting (a generally real-parallelism application)[39] to game-tree searching[24] (generally speculative) have been written and analyzed under various MULTILISP environments. However, a feature missing from the language becomes clear when writing speculative codes; since *FUTURE* is the *sole* function for specifying parallelism in any way, there is no general methodology for *controlling* that parallelism.

Chu[24] shows ways to implement communication between parallel MULTILISP processes, and suggests that codes use these pathways to communicate messages about process termination and so on. Either explicit code to handle termination messages, or some exception signalling, would then be used to remove speculative processes from the execution environment. However, although some work on exception handling has been published for MULTILISP, no complete semantics are given for exception propagation in the language.[40] However, new work in speculation control for MULTILISP implementations is proceeding at this time.[60]

Gabriel and McCarthy's QLISP takes a different tack to specifying parallelism within LISP.[35, 36] They choose the COMMON LISP dialect for augmentation, with a half-dozen parallelism specification and control constructs thrown in. The first new QLISP construct they introduce is **qlet**:

```
(qlet control
      ((x (compute-x))
       (y (compute-y)))
  (combine x y))
```

The **qlet** construct allows programmer specification of parallelism under dynamic control. In particular, in the above code, if the value of **control** is *nil*, then the entire form acts exactly as it would if it had been written as a standard COMMON LISP **let** form (*i.e.*, sequentially compute values for **x** and **y** and call the function **combine** on the results). If, however, the value of **control** is non-null, then processes are created to execute the

functions **compute-x** and **compute-y**; these processes may concurrently execute, after which **combine** is called on the results. This is very much like the MULTILISP addition example we saw previously with an small difference: the function **combine** is not executed until the processes computing **x** and **y** have completed.

However, QLISP allows both behaviors. If the value of **control** happens to evaluate to the symbol **eager**, then the function **combine** is started simultaneously with the parallel computations of **x** and **y**. The values for **x** and **y** passed to **combine** are simply placeholders exactly analogous to MULTILISP's *futures*.

This may be a useless step to take if **combine** relies heavily on its inputs, as did the + function in the MULTILISP addition example. However, if **combine** can find something else to do, some more parallelism may be gleaned from the system. For example, in the code

```
(defun combine (a b)
  (expensive-computation)
  (+ a b))
```

the **expensive-computation**, which does not reference the arguments **a** and **b** of **combine**, might complete some work while the caller of **combine** above spawns processes generating values for the arguments. QLISP also includes explicit functions for introducing parallel queue-processing functions (*i.e.*, sequential processes communicating via queues *à la* communicating sequential processes[45]); hence the "Q" in QLISP.

In addition, QLISP provides for explicit process termination, and allows "remote" termination (*i.e.*, one process terminating another) via the communication queues implicit in the parallelism model. The termination control is built around analogs to the COMMON LISP **catch** and **throw** nonlocal transfer-of-control functions; these functions are given process-termination semantics as well.

Unfortunately, the specification of parallelism, whether speculative or real, is still entirely up to the programmer. In addition, no hints are given the prospective implementor of QLISP as to how to go about implementing the unclear semantics of process termination and context switching in the language. QLISP specifies that the address space of all processes is shared; what about the special (dynamic) bindings of variables, a feature inherited from COMMON LISP? If bindings are not kept on a per-process basis, the language might display

unwanted nondeterminism; however, implementing dynamic binding in another fashion may be expensive. How is the state of a process "wound up" after termination via **throw**? These and other implementation (and detailed semantics!) questions need answering.

Another well-studied model of parallel computation is the *Actor* model of Hewitt and others. The ACT3 language (and its predecessors ACT and ACT2) have been studied for years at M.I.T. In ACT3, both real and speculative parallelism may be specified explicitly by programmers in a communicating sequential processes style. Messages between *actors* are queued in *mailboxes*, much as in the queues in QLISP. Processes may be created (by an explicit **create** operator) and mutated (by the **become** operator) to have specific new behaviors; control over processes must be communicated by the same channels as any other messages. The termination of processes is not explicitly supported, although programs may be written to recognize certain messages as requests to stop using system resources, and then **become** an actor which does nothing. Actor-model researchers note that programmers need not be explicit about opportunities for parallelism in their ACT3 codes, but must *"concentrate on thinking about the parallel complexity of the algorithm used."*[2] Thus the parallelism of the application must *still* be programmer-specified.

The ETHER language described by Kornfeld[49] is a language with LISP syntax. This interesting variant of LISP allows implicit creation of speculative processes within dynamically enclosing knowledge bases with some automatic inference based on programmer-specified rules. The function **activate** is used to initiate a parallel process created by **new-activity**:

```
(defun solve-problem (list-of-alternatives current-knowledge)
  (foreach possibility list-of-alternatives
    (activate (new-activity)
      (if (constitutes-solution? possibility)
          (within-viewpoint current-knowledge
            (assert (answer-is →possibility)))))))
```

The function **solve-problem** above could be used to check each element of a list (with the function **constitutes-solution?**) to see if it solves a given problem. The result is returned to the caller by using **assert** to change the caller's knowledge base. ETHER also includes primitives to control tasks, including a termination function **stifle**. The call

```
(stifle activity)
```

explicitly stops the execution of the process bound to **activity**. Unfortunately, again we have a language requiring explicit programmer control over task termination, and an incomplete understanding of the effects of termination on the semantics of the language.

The A. I. literature is quite full of various methods for implementing parallel execution of PROLOG.[†] As a parallel search specification language, PROLOG is quite useful, since in effect each rule application of a rule with multiple clauses (*i.e.*, nearly all rules) is a choice point in a decision tree. Although some researchers have concentrated on this approach to taking advantage of the implicit *OR*-parallelism in PROLOG programs,[3] other approaches are generally more popular due to the space overhead of the environment copying required by *OR*-parallel execution of PROLOG programs. Nevertheless, these approaches allow implicit parallelism (*i.e.*, not explicitly programmed-specified). However, again termination semantics are unclear for the control of the rapidly growing search tree which results from this style of execution.

Several authors have researched variants of PROLOG which allow some automatic and some explicit control of the other parallelism that can be found in PROLOG programs (*i.e.*, *AND*-parallelism of multiple goals to be proved in proving a clause).[30, 75] This again leads us back to programmer-specified parallelism.

Another interesting class of A. I.-inspired parallelism research is the massive parallelism approach. Typified by languages such as NETL[34] and architectures such as the Connection Machine[44], massive parallelism approaches (also called *active memory* approaches) generally try *all* possible answers for a problem, and simply waste computation power. For example, in a database search, a Connection Machine could be configured such that each element of the machine represented a database record; a search then proceeds as a parallel match between the query and every record, with each processor checking its associated database record. Assuming only one match is found, the rest of the computation is "wasted." Despite this limitation, there are a large class of speculative algorithms that are

[†]This is actually a contradiction in terms, because of PROLOG's sequential semantics. We mean instead a parallel implementation of a language with PROLOG's basic syntax, but some different (and parallel) semantics.

well matched to this approach, such as database searches, certain vision algorithms, *etc.*

Despite various approaches to allowing, controlling, and maintaining parallel search, for speculative problems researchers will always attempt to pare the search space as much as possible. However, the current activity in architectures for parallel A. I. programming show that even after limiting the search space as much as possible, A. I. programmers still find they could take advantage of some ability to specify speculative parallelism. There are problems in the A. I. literature, for example, that are known not to have polynomial-time solutions, yet which are still useful to solve in a search-like manner (a perfect example is the provably co-NP-complete subsumption problem in knowledge representation[21]).

Therefore, this report will attempt to allow speculative parallelism within the ID language and under dataflow paradigms of execution. Our goal is to avoid any explicit programmer specification and control of parallelism, either real or speculative (as is currently true in ID) and yet maintain the useful invariants of the ID language and the underlying dataflow instruction firing mechanism (such as self-cleaning dataflow graphs). We will look at a small set of search problems that are simple to specify, yet form the core of (or are similar to) many A. I. codes. We choose a class of search problems in which any solution among a set of solutions forms an "acceptable" answer (*i.e.*, even if more than one answer to a problem exists, a single answer is an acceptable result). This still forms a useful class of programs, particularly for game-tree search—it doesn't matter how we win, as long as we win!

1.5 Road Map

The balance of this report presents solutions to embedding controlled speculation in the ID language and the tagged-token dataflow architecture in a consistent, programmable manner. Chapter 2 discusses language syntax and semantics changes in the ID language to support these new features, with examples of usage and motivations for the approach.

Chapter 3 goes on to present implementation strategies for these new structures within the ID language. Approaches are presented for each, with advantages and disadvantages clarified. Close scrutiny is given to the general approach to resource control in the ID language, and how it relates to controlling speculative usage of resources.

Chapter 4 expands on these structures by presenting the architectural support for these ID

language structures within dynamic dataflow execution paradigms. The abstract tagged-token dataflow architecture[9, 11] is taken as the starting point for architectural extensions. Appendix A clarifies the semantic requirements of dataflow operators in order to properly support speculation control, and gives compiler schemas for various major ID language constructs based on those semantics.

Chapter 5 presents a finished application written in ID and using the new speculative constructs developed in the report. Some commentary is also presented on the results of executing this application. Measurements of the success of the approaches are discussed. The final chapter, Chapter 6, summarizes the impact of the constructs of this report on the ID language and the tagged-token dataflow architecture. Suggestions for further research and development are also presented.

In testimony before a congressional panel investigating the causes of the 1929 Crash, Bernard Baruch was asked to describe his occupation. He replied: "a speculator."

— A. MICHAEL LIPPER, *Back to the Future: The Case for Speculation, Baruch-Style*

Chapter 2

Language and System Design

We propose to extend the syntax and semantics of the ID language in order to fully support controlled speculation at an abstract level. The approach to controlling speculative parallel execution that we take involves nondeterministic constructs and other features that are not currently defined within the ID language. We begin by justifying these large changes in ID language semantics.

2.1 Controlling Speculation

The fundamental problem we are trying to solve is taking advantage of the speculative behavior of many programs to provide another source of parallelism from which dataflow compilers and machines may benefit. Other methodologies for automatically handling speculation "lazily" exist, even under dataflow execution paradigms[43, 42]. It has even been shown that such approaches are optimal in the total amount of work done.[20] Baker and Hewitt point out, however, that lazy evaluation is not optimal for multiprocessors, in the sense that it does not necessarily generate the shortest critical path.[15]

Therefore we take the approach of allowing parallel execution of speculative branches of program choice points. However, the parallelism generally found in speculative programs has a tendency to be explosive; it therefore must be controlled in some fashion not currently explored in dataflow research.

The search for methods of control is aroused by the seemingly obvious source of parallel execution. Even simple algorithms can exhibit large amounts of this speculative parallelism.

For example, the standard minimax algorithm is used to find a best-path traversal of trees of data. In a game tree (representing the possible moves and counter-moves of a competitive game), each tree node would contain an estimate of the relative probability, on some well-ordered scale, of winning the game given a certain sequence of moves from that point on. A program which is evaluating the possible moves from the current game situation will choose a move which maximizes this probability of winning.[†]

The usual implementation of this algorithm models the possible moves of a game as a tree. Each node of the tree represents a possible sequence of game moves from the current situation; the branching factor of each node is equal to the number of legal moves after a given sequence of moves. The root of the tree represents the current game situation. Finally, the leaves of the tree represent the relative merits of all possible "final moves" in the game.

For all interesting games, the size of the tree just described would be immense. The game of chess, for example, presents a branching factor of up to 32, with games consisting of over 100 moves. Clearly the computational expense of searching the entirety of this tree is too high; thus most game programs limit their search space by looking ahead some set number of moves (plies). The values at the leaf nodes, therefore, represent some approximation of the probability of winning the game after reaching that node.

The minimax procedure assumes that both players wish to win the game, and thus traces the sequence of moves which maximizes this probability *for each player*. That is, at levels of the tree representing *our* moves, we choose the *most* advantageous move for us; at the levels of the tree representing the *opponent's* moves, we choose the *least* advantageous move for us. Thus the name *minimax*, for the successive maximizing and minimizing, arises.

For example, in the minimax tree in Figure 2.1, we have a three-ply lookahead into some game. The best sequence of moves we can find from the current starting point is the move signified as the left branch, as this leads us to the best result (eight) of the minimax search.

The algorithm to compute this search is given below coded in ID:

[†]There is an implicit assumption in this approach of "rational play," *i.e.*, that the opponents are both trying to win and will always find the best move in each situation. This does not, however, impact our discussion of speculation and speculative parallelism; see Nilsson[59] for more information and references.

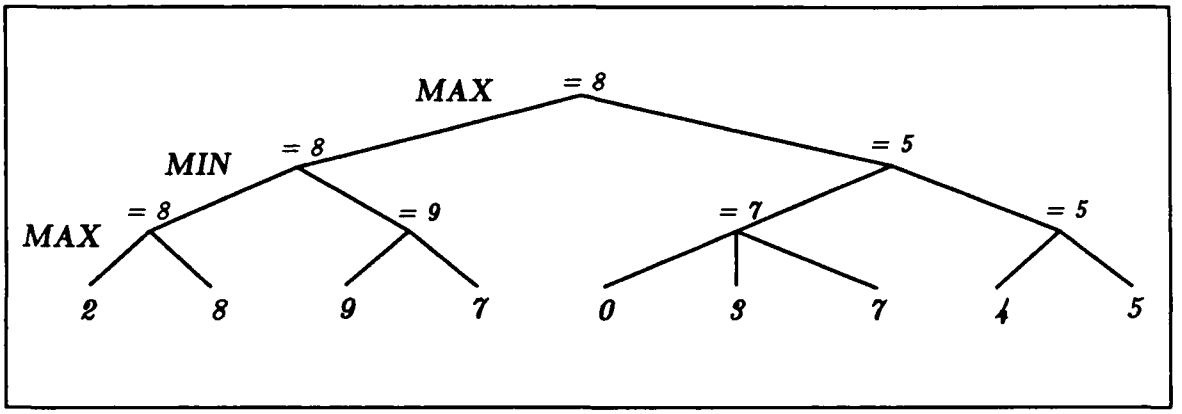


Figure 2.1: Minimax Game Tree

```
type tree = leaf N
|           subtree (list tree);

def minimax node = mm (max, min) node;

def mm (maximize, minimize) (leaf value) = value
|   mm (maximize, minimize) (subtree nodes) =
        fold maximize (map_list (mm (minimize, maximize)) nodes);
```

Ideal simulation of this straightforward algorithm applied to a three-ply (depth three) tree with a branching factor of ten showed maximum ALU (non-overhead) parallelism levels of 1,361 searching all 1,111 nodes in the tree. The average parallelism fared better than 354, as can be seen in Figure 2.2. The critical path of the simulated execution was only 360 instructions long. This is the shape of the parallelism profile we would expect, with a critical path of length $O(d)$ and maximum parallelism $O(b^{d-1})$ for a search tree with a branching factor of b and depth of d, since there is no search constraint between subtrees of a node.

This result is not realistic, however. In any real machine, we would constrain the parallelism arising in the **map_list** function to match the real available machine power. This approach to parallelism control is called *loop bounding*, and is fully explored in other work.[28] For the rest of this discussion we will present statistics which imply that no loop may be "unrolled" more than an arbitrary six instances at a time (for purposes of comparison), using this loop bounding approach.

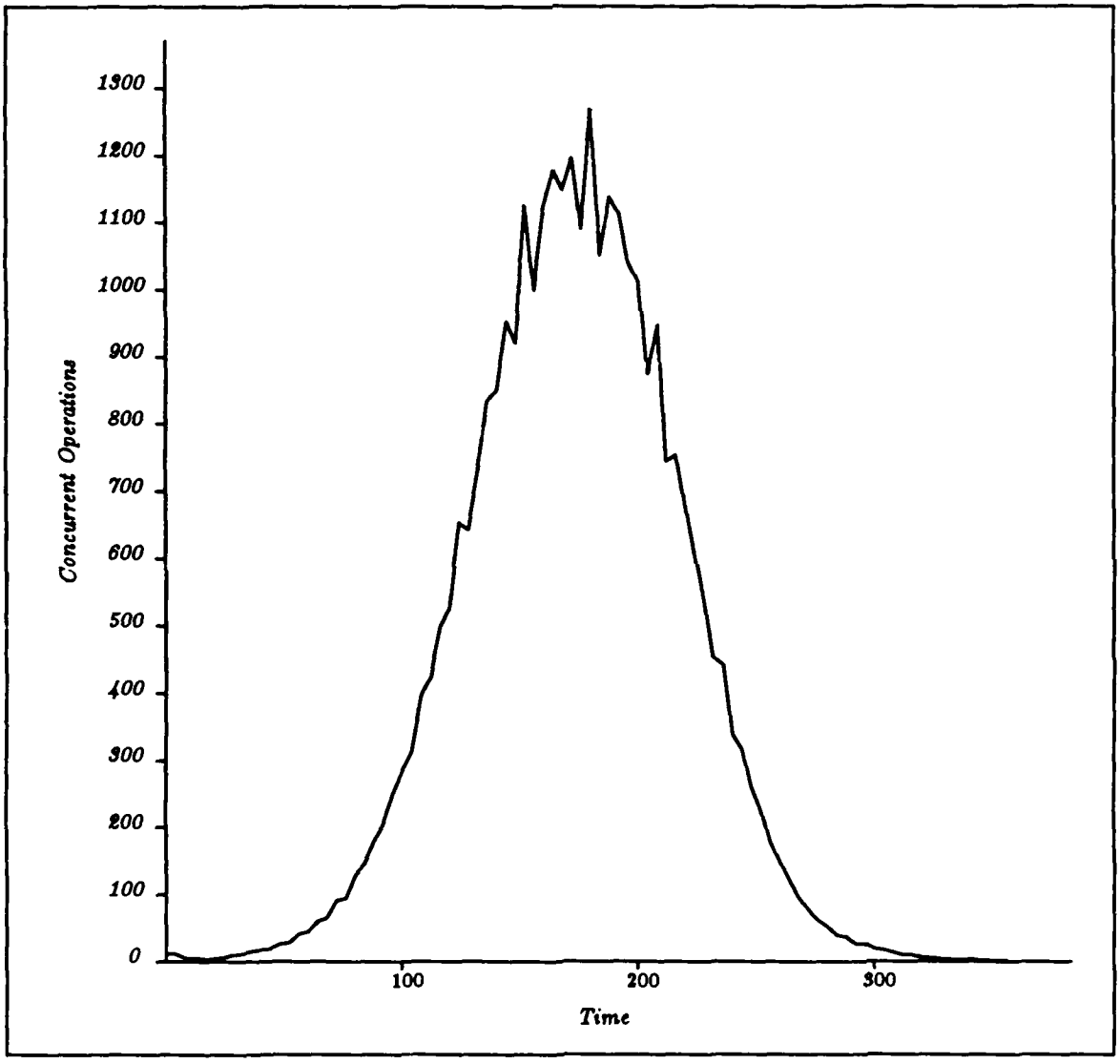


Figure 2.2: Minimax Algorithm: ALU Parallelism Over Time

Re-executing the same minimax code with the same input data and a loop bound of six yields somewhat different results. The same number of instructions were executed (127,652), but the critical path was more than doubled to 787 as portions of the execution tree "slid" later. The maximum ALU parallelism dropped some 69% to 427, with an average parallelism of about 162. Figure 2.3 shows this ALU parallelism over execution time.

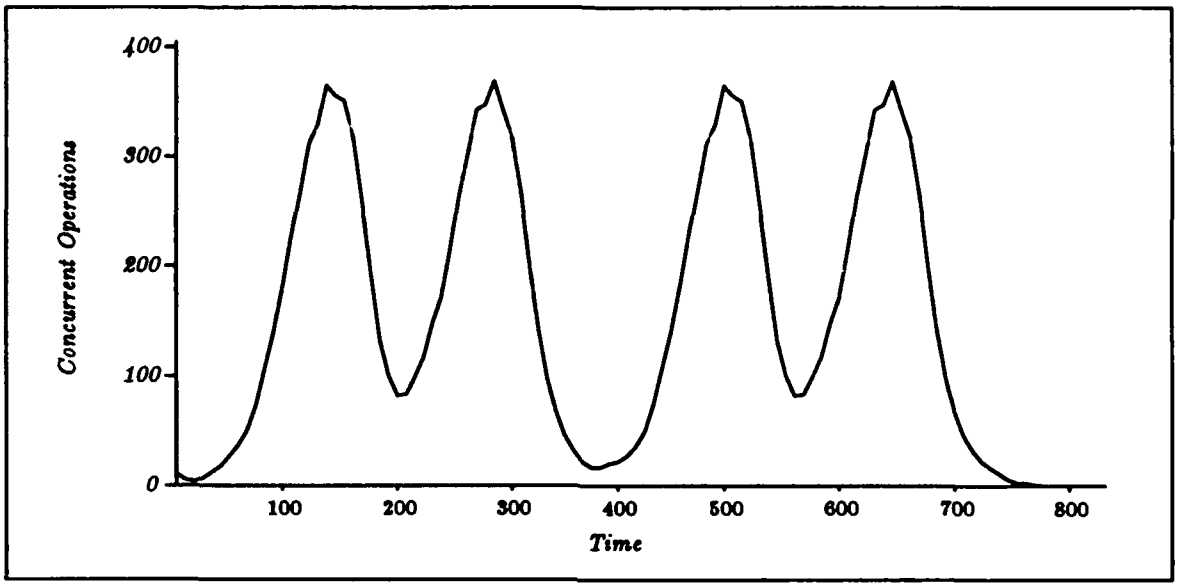


Figure 2.3: Minimax Algorithm with Bounding: ALU Parallelism Over Time

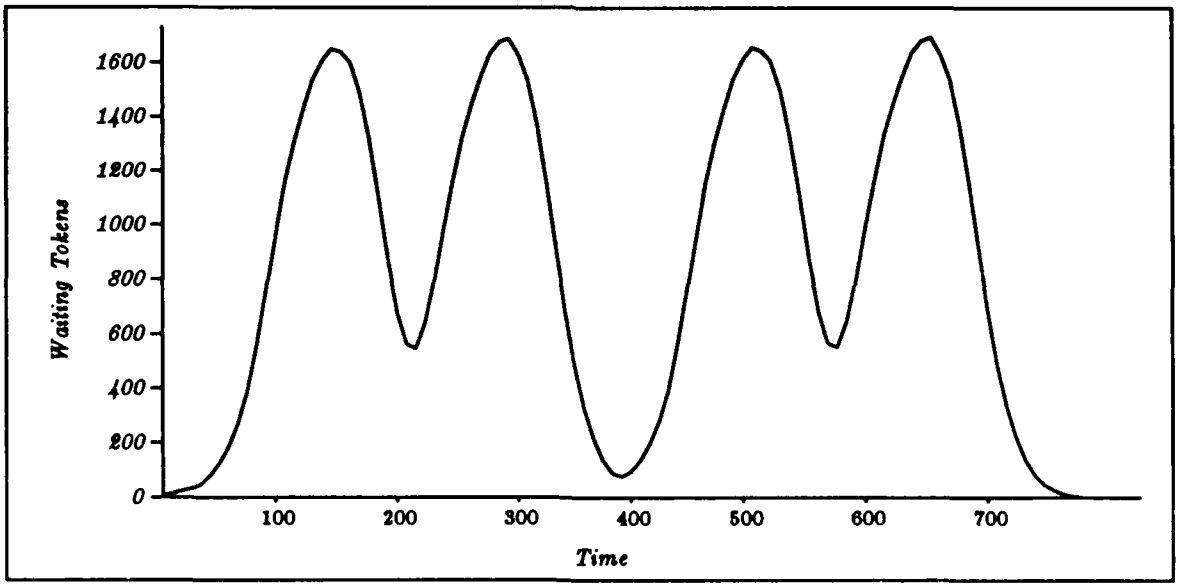


Figure 2.4: Minimax Algorithm: Token Storage Over Time

Importantly, however, the loop bounding approach to loop control has lowered the system

resources necessary to store pending execution tokens. While the unbounded execution required token queues of length 5,734 to store pending operations, the bounded execution lowered this number to 1,700, or 69% smaller token queues. The profile of token storage needed over execution time is in Figure 2.4.

The loop bounding certainly controlled the unbridled indulgence in system resources. However, *exhaustive* speculation is what we indulged in; even in the bounded scheme, we searched all 1,111 nodes of the search space (which accounts for the instruction counts being the same). Better search methodologies exist for searching minimax trees.

2.2 Serialization is Used to Control Speculation

Many search algorithms rely on serialization (sequentialization) to control this speculation in finding "correct" or "best" answers. A good example is the well-known *alpha-beta* search algorithm[59, 62], which relies on knowledge of already-searched portions of the search tree to help "prune the search path," that is, ignore portions of the search tree which can be proven to be fruitless.

The alpha-beta search pruning paradigm, therefore, relies on some particular ordering (for example, depth-first) and later knowledge of previous results from the search. With this knowledge, we can ignore portions of the tree which can be proven to be irrelevant to the final minimax result. For example, in Figure 2.5, the tree outlined in Figure 2.1 is searched by the alpha-beta minimax procedure. Under this procedure, the third leaf node (containing nine) implies that the result of minimax search on its parent node will be at least nine (no less, since the parent is a maximizing node); however, the parent of *that* node has been shown to be eight or less. Therefore, we needn't search more descendants along this path. A "deeper" cutoff (*i.e.*, containing a larger subtree) can be seen on the right of the figure, in which we can ignore an entire subtree.

In ID, this algorithm can be outlined as follows:

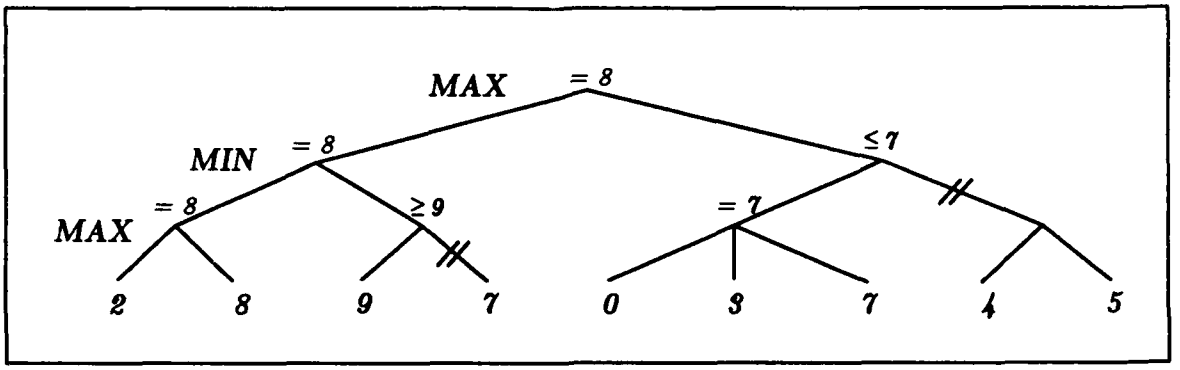


Figure 2.5: Minimax Game Tree with Alpha-Beta Cutoff

```
def alphabeta node = ab ((<), (>)) (−∞) ∞ node;

def ab (lesser?, greater?) α β (leaf value) = value
|   ab (lesser?, greater?) α β (subtree nodes) =
      { def minmax x y = if lesser? x y then y else x;
        value = α
        in
          { while nodes ≠ nil and lesser? value β do
              node : next nodes = nodes;
              next value = minmax value (ab (greater?, lesser?) β value node)
            finally value }};
```

This algorithm, as can be seen from Figure 2.6, is quite sequential, since it simply examines one node at a time. Nevertheless, the pruning technique did reduce the total number of instructions executed by nearly 55% to 57,604, by searching only 346 nodes of the search space. In fact, in has been shown by Knuth[47] that the alpha-beta search paradigm in general explores $O(\sqrt{n})$ nodes of an n-node search tree, if all move choices are ordered best-first. Therefore, the benefits of alpha-beta search increase with tree size. We expect to see, and the shape of the parallelism profiles do show, a maximum parallelism of $O(1)$. The critical path, however, has lengthened from the ideal minimax $O(d)$ for a tree of branch factor b and depth d to

$$
\begin{aligned}
O(\sqrt{n}) &= O\left(\sqrt{\sum_{k=0}^{d-1} b^k}\right) \\
&= O\left(\sqrt{\frac{b^d - 1}{b - 1}}\right) \\
&\approx O\left(b^{\frac{d-1}{2}}\right)
\end{aligned}
$$

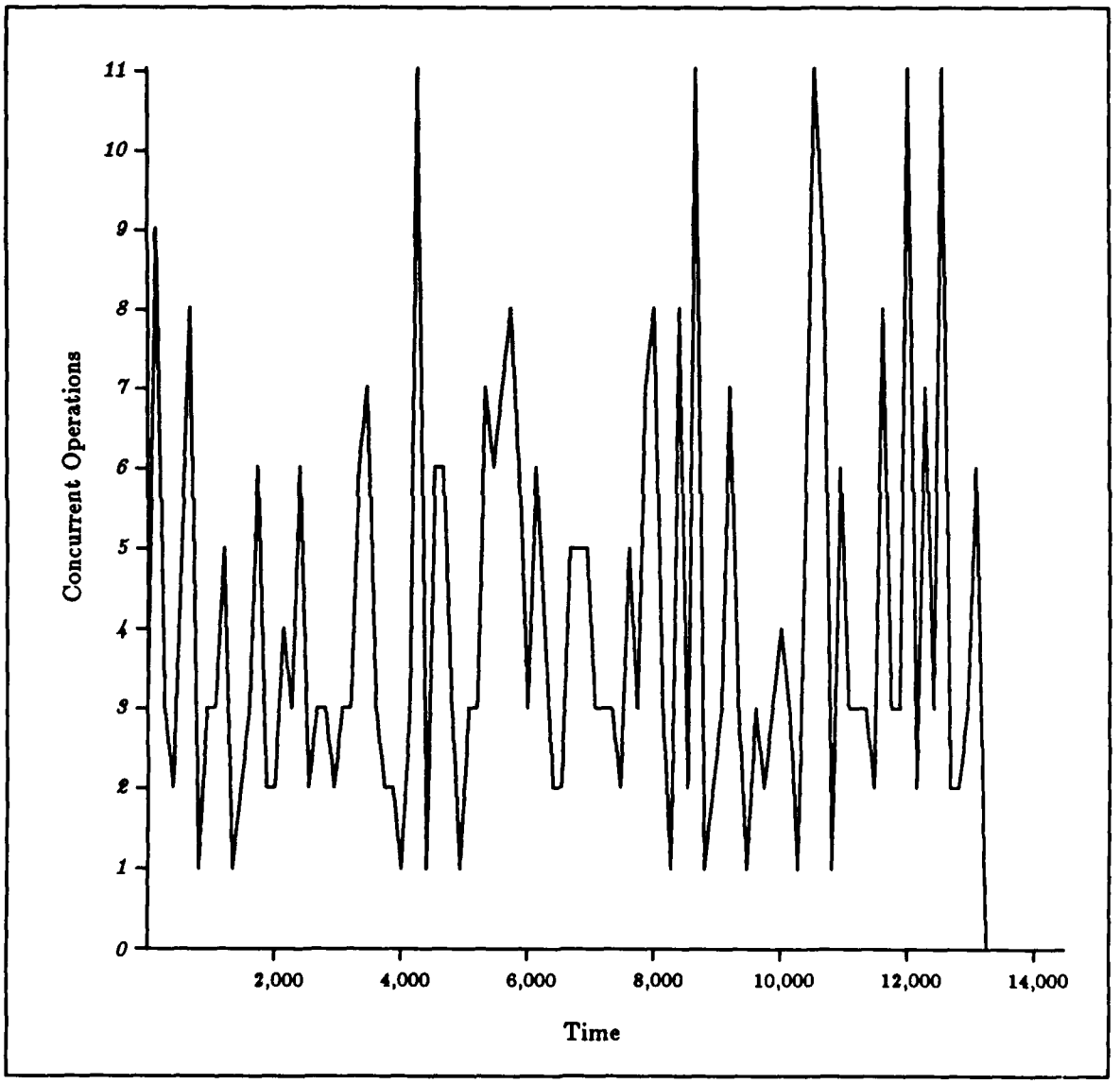


Figure 2.6 Alpha-Beta Algorithm: ALU Parallelism Over Time

In other words, in this particular case, these cutoff gains were at the expense of a critical path now 17 times longer! Average parallelism was only 4.3, peaking at eleven, with maximum token storage requirements only 86 tokens. Although a large segment of the search space is

ignored by virtue of the alpha-beta approach, nearly no parallelism is found in what seems intuitively to be (and clearly was, under the minimax algorithm) a parallel problem.

2.3 A Hybridized Search Scheme

A middle ground is needed to bridge this gap. If it was possible to support some communication between the subscarches of the choice points of a node of a search tree, we might be able to prune parts of a search in a manner equivalent to the alpha-beta search pruning technique. We propose to extend the heretofore determinate ID language with constructs that are not only non-functional in nature, but present a nondeterministic primitive functionality to the ID programmer.

The following code will search alternatives within a game tree *in parallel*, nevertheless allowing pruning of the search tree, using imperative-style writing into I-structure cells. The **blackboard** primitive is used to create such a communications cell; the ":=" construct is used to write into such a cell imperatively (*i.e.*, without the benefit of I-structure presence detection). We emphasize that this proposed construct is a nondeterministic *extension* to the ID language.

```
def hybrid node =
    { α = blackboard ∞;
      β = blackboard (−∞);
      in
         h ((<), (>)) α β node };

def h (lesser?, greater?) α β (leaf value) = value
|   h (lesser?, greater?) α β (subtree nodes) =
    { def minmax x y = if lesser? x y then y else x;
      value = α[0];
      child = blackboard value
      in
         { while nodes ≠ nil and lesser? child[0] β[0] do
              node : next nodes = nodes;
              next value = minmax child[0] (h (greater?, lesser?) β child node);
              child[0] := value     % Write value onto the "child" blackboard.
           finally value }};
```

The **blackboard** construct represents a communication area which branches of the search

tree can consult to ascertain whether cutoff should occur. The "blackboarding" communication paradigm is borrowed from the A. I. parallel-programming literature;[33] the analogy is to a team of people cooperating on the solution to a problem, using a blackboard to store subproblem results which may be useful to other members of the team. In this case, as the search progresses, the blackboard for each node of the search tree reflects an incrementally refined "best yet" result which might allow alpha or beta search cuts of subtrees. In effect, we can think of the search at each point as a stepwise refinement of the correct alpha or beta value at that choice point, with subnodes noting that value at various levels of refinement. The alpha or beta value seen by a subtree of a choice point can be thought of as a *temporal approximation* of the correct alpha/beta value, *i.e.*the best value found *so far.*

The semantics of the **blackboard** construct can be outlined in a rewrite manner similar to that used in describing the remainder of the ID language.[8] In this approach, we specify the execution of an ID program as a series of reductions to the final result. The following program fragment demonstrates the semantics of the **blackboard** and := constructs:

{ B = **blackboard** 1;
  B[0] := 3;
  B[0] := 5
  in
    B[0] }

$\Longrightarrow$ B[0]
- B = **blackboard** 1;
- B[0] := 3;
- B[0] := 5

$\Longrightarrow$ <b0>[0]
- <b0>0 := 1
- <b0>0 := 3
- <b0>0 := 5

$\Longrightarrow$ b0
- b0 := 1
- b0 := 3
- b0 := 5

$\Longrightarrow$ 1 *or* 3 *or* 5

We have extended the reduction meta-language of Arvind and Ekanadham to include the

or operator, specifying nondeterministic choice of several results. Thus the result of *reading* a blackboard object (*i.e.*, referencing a blackboard such as in B[0]) at any given time t_n has the semantics of *one of* the values *written* to the blackboard at some time $t < t_n$. There is no guarantee of *which* value will be returned, which gives us just the small amount of nondeterminism we needed to unchain the strict sequentiality of the alpha-beta search formulation.

Using these new constructs, we can see in Figure 2.7 the parallelism of the hybrid approach (in this particular execution, with the same game tree). The maximum ALU parallelism is now 325, with a critical path of 5,359. 822 nodes of the search tree were explored, more than twice the number as were searched in alpha-beta search, but only 74% of the total nodes searched using the simple minimax approach. The maximum ALU parallelism was pared from simple minimax by nearly 24%, while the maximum token storage needed was cut from the simple minimax by 20%. However, the hybrid method displays a critical path of 5,350, a full 60% shorter than the fully sequentialized alpha-beta search.

We can see a pronounced difference in storage and processing requirements in the hybrid approach. Figure 2.8 shows the maximum number of function invocations executing at each time step of the execution of the program along the critical path. As expected, the minimax procedure finishes first and presents the most parallelism (much of it unnecessary). The alphabeta procedure takes the longest to execute (by far), but does not exhibit much parallelism. The hybrid approach seeks out the happy medium.

It is important to note that nondeterministic primitives such as **blackboard** and := do introduce an element of surprise. Even the simple program above can exhibit different resource usage behavior on every multiprocessor execution, due to its nondeterminism. However, *the program will always return the same result, regardless of execution order and resource usage.* This is an important property, enforced only by the disciplined ("conventional") use of nondeterministic constructs by the programmer, and *not* by any Church-Rosser properties of the language itself.[†]

The ID language, as augmented by the **blackboard** construct, does not in fact exhibit the

[†]And just to prove the difficulty of programming with explicit nondeterministic constructs, the careful reader will notice that the hybrid alpha-beta code presented contains a race condition between the **minmax** computation and the writing of **value** into **child** which may cause incorrect results. We will ignore this problem for now, as it does not directly impinge on our discussion of the blackboard approach. However, this problem will become more grist for our argument of encapsulated speculation controls later on.

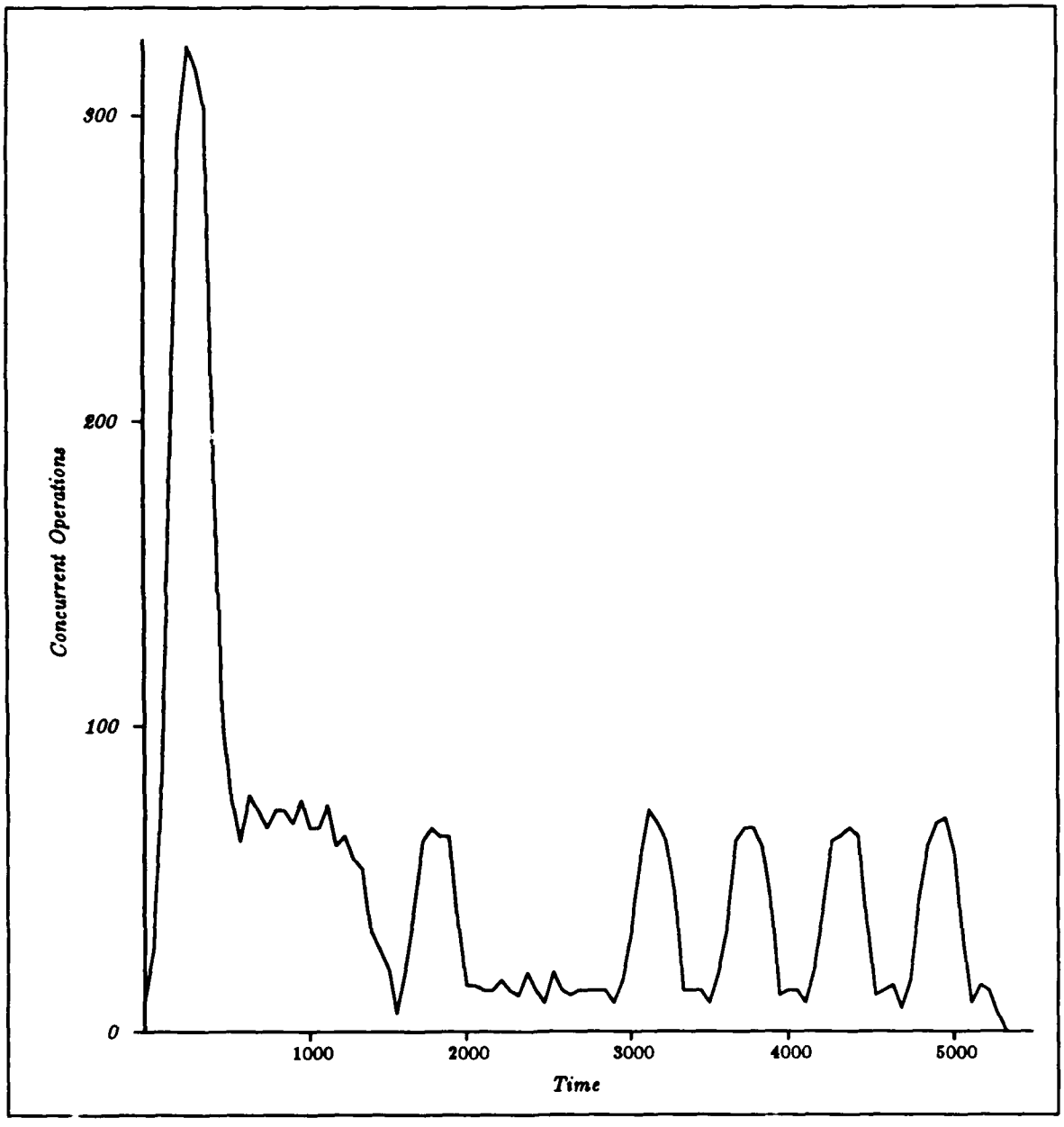


Figure 2.7: Hybrid Minimax Algorithm: ALU Parallelism Over Time

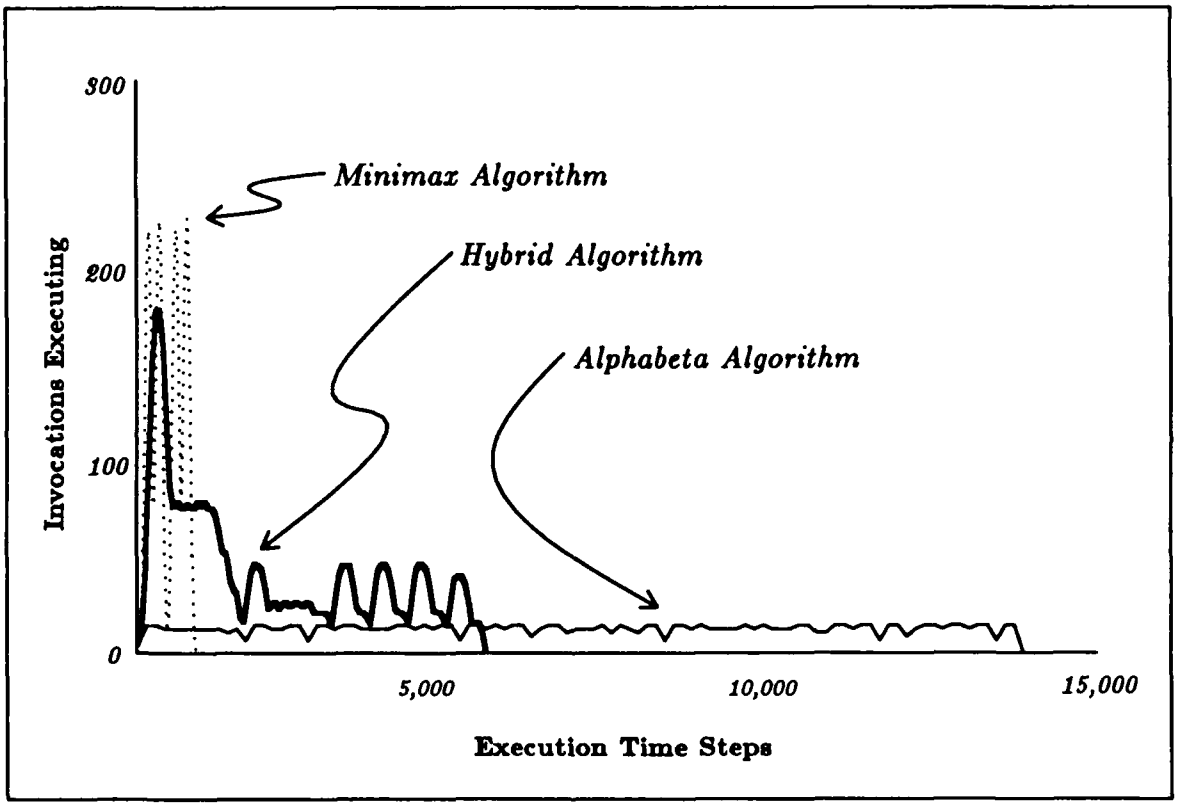


Figure 2.8: Invocations Executing at Each Time Step for the Three Algorithms

Church-Rosser property[29]; execution order does (in general) affect the result of nondeterministic computations. Instead, the determinacy of the result follows from programmer-enforced consistency. The correctness thus exhibited is akin to the *sequential consistency* developed by Lamport[51]. We are depending on the nature of the problem, in particular the conservative cutoff criterion of the hybrid alpha-beta algorithm, to enforce this invariant.

2.4 A Concrete Example: A Cryptarithmetic Solver

Let's choose a specific example on which to explore extensions of this approach to parallelism control. The ID code presented below solves typical "cryptarithmetic" problems as presented by Kornfeld in discussing parallel speculation.[49] A cryptarithmetic solver finds a solution to the class of "mind bender" problems in which ten alphabetic characters are assigned digit values from zero to nine, such that the addition of two strings of characters results in another particular string.[†] An example problem is outlined in Figure 2.9. In this example, we seek a matching of the characters (D O N A L G E R B T) to the numbers from zero through nine which makes the addition correct.

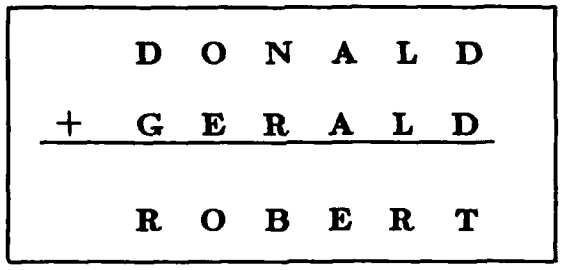


Figure 2.9: Typical Cryptarithmetic Problem

Such problems have "obvious" search trees of size 10!, though of course much pruning can be applied. This class of problems is particularly interesting, as input order and search order can have a profound impact on the number of solution nodes which must be searched to find a correct answer. In one particular solution algorithm for solving the problem in Figure 2.9, the same program that solved the problem by searching only 3,988 nodes had to search 1,995,840 nodes to find the *same answer* when presented the problem in a different order.

[†]Typically, to make such mind benders more interesting, the three strings of characters form English words, often proper nouns.

The following code implements the cryptarithmetic solver in a depth-first manner. The core of the solver is in the recursive **solve_crypt** function, which controls the search. In actual usage, the **crypt** function would be presented with three lists of symbols representing the addends and result strings of the problem.

```
% Top-level: take the three lists, generate a merged symbol
% search order and present to solve_crypt.
def crypt a1 a2 a3 =
    { r1 = reverse a1;
      r2 = reverse a2;
      r3 = reverse a3
      in
        solve_crypt r1 r2 r3 (merge r1 r2 r3) nil };

% Solve a crypt puzzle recursively. Check if a solution has
% been found or is impossible; else recursively try all possible
% bindings for the next letter in the search list.
def solve_crypt l1 l2 result letters bindings =
    if not (check_solve l1 l2 result bindings)
      then nil
      else if letters == nil
             then bindings
             else { letter : rest = letters;
                    _, digits_used = unzip2 bindings;
                    def try nil = nil
                    |   try values =
                        { v : vt = values;
                          answer = solve_crypt l1 l2 result rest
                                                ((letter, v) : bindings)
                          in
                            if answer ≠ nil
                              then answer
                              else try vt }
                    in
                      try (difference (0 to 9) digits_used) };

% Subtract sublist from list set-wise.
def difference list sublist =
    { result = nil
    in
      { for item ← list do
            next result = if member? (==) item sublist
                            then result
                            else item : result
        finally result }};
```

%%% This code is continued on page 42.

%%% *This code is continued from page 41.*

```
% Merge three lists in an order good for search. The search
% order we choose is from low-order to high-order digits, so
% that chances for search cutoff are magnified.
def merge s1 s2 s3 =
    { result = nil;
      def glue nil s2 s3 = nil
      |   glue s1 s2 s3 =
          (hd s1) : (hd s2) : (hd s3) : (merge (tl s1) (tl s2) (tl s3))
      in
         { for item ← glue s1 s2 s3 do
               next result = if member? (==) item result
                             then result
                             else item : result
           finally reverse result }};

% Check bindings to see if we have a possible solution.
def check_solve l1 l2 result bindings =
    { def check nil l2 result carry = carry == 0
      |   check l1 l2 result carry =
          { v1 = lookup (hd l1) bindings;
            v2 = lookup (hd l2) bindings;
            vr = lookup (hd result) bindings
            in
               if (v1 < 0) or (v2 < 0) or (vr < 0)
                 then true
                 else { total = v1 + v2 + carry
                        in
                           if (mod total 10) ≠ vr
                             then false
                             else check (tl l1) (tl l2) (tl result)
                                        (floor (total/10)) }}
      in
         check l1 l2 result 0 };

% Look up a symbol's binding in current environment.
def lookup symbol nil = -1
|   lookup symbol bindings =
    { (s, v) : rest = bindings
      in
         if s == symbol
           then v
           else lookup symbol rest };
```

This solution of the problem uses knowledge of partial addition to decide during the search whether portions of the subtree need not be searched. This is accomplished by calculating as much of the puzzle as possible with the unfinished solution. For example, in the $DONALD + GERALD = ROBERT$ problem presented, if in any portion of the search tree we have $D + D \bmod 10 \neq T$ then the current and all future search subtrees may be terminated, as they cannot possibly bear fruit.

Using this knowledge, the **crypt** function orders the search (using the **merge** function) such that partial solutions can be checked for correctness. Then possible bindings for each letter in turn are considered by speculation in the recursive **solve_crypt** function. This function then uses the **check_solve** (and helper function **lookup**, for looking up extant bindings for symbols) to check for the impossibility of a set of bindings, which should terminate that branch of the search tree.

As was the case in the alpha-beta tree search, this solution is quite sequential, as prior knowledge of previous search paths is required before the search continues down another path.[†] However, there is no *a priori* knowledge that one particular search path is better than another at any given point; all possibilities are equally likely to be solution paths. Therefore we might take the liberating approach of a parallel speculative approach, by reimplementing **solve_crypt** to search all possible paths concurrently:

[†] *I.e.*, this code implements a depth-first search.

```
% New version of solve_crypt removes the if serialization
% previously found in the try function, allowing parallel
% search of all subtrees of a choice point.
def solve_crypt l1 l2 result letters bindings =
    if not (check_solve l1 l2 result bindings)
      then nil
      else if letters == nil
             then bindings
             else { letter : rest = letters;
                    _, digits_used = unzip2 bindings;
                    def try nil = nil
                    |   try values =
                        { v : vt = values;
                          answer = solve_crypt l1 l2 result rest
                                                ((letter, v) : bindings);
                          another_answer = try vt
                          in
                             if answer ≠ nil
                               then answer
                               else another_answer }
                    in
                       try (difference (0 to 9) digits_used) };
```

As expected, there is a distinct critical-path shortening from the sequential to the parallel version (nearly six times); yet we find an even worse explosion of work in the parallel version (more than *sixty* times). Again we might generate a hybridized version using the **blackboard** approach; in this code, each **blackboard** holds either *true* (meaning that a solution has been found at the previous search level) or *false* (a solution has not yet been found). The **solve_crypt** procedure checks the blackboard **done** before doing any work; likewise, it writes *true* into the cell passed down to inferior searches (**cell**) when an answer has been found.

```
% Hybridized cryptarithmetic solver, using blackboards.
def crypt a1 a2 a3 =
    { cell = blackboard false;
      r1 = reverse a1;
      r2 = reverse a2;
      r3 = reverse a3
      in
        solve_crypt r1 r2 r3 (merge r1 r2 r3) cell nil };

% New version of solve_crypt without the if serialization
% but with cutoff controls using blackboard.
def solve_crypt l1 l2 result letters done bindings =
    if done[0]
      then nil
      else if not (check_solve l1 l2 result bindings)
             then nil
             else if letters == nil
                    then bindings
                    else { letter : rest = letters;
                           _, digits_used = unzip2 bindings;
                           cell = blackboard false;
                           def try nil = nil
                           |   try values =
                               { v : vt = values;
                                 answer = solve_crypt l1 l2 result rest cell
                                                      ((letter, v) : bindings);
                                 another_answer = try vt
                                 in
                                   % Check to see if we're done.
                                   if answer ≠ nil
                                     then { cell[0] := true % Yes, notify.
                                            in
                                              answer }
                                     else another_answer }
                         in
                           try (difference (0 to 9) digits_used) };
```

This slightly more complex code, like the hybridized alpha-beta search previously seen, will perform some cutoff of search paths once an answer has been found, *provided that the other subtrees of a choice point have not yet started.* This is a crucial point; since although cutoff is *detected* (and recorded) at all levels of the search, it is only *checked* for one level deep (inside **solve_crypt**). Thus, even though a complete solution has been returned to **crypt**, we might find that search subtrees many levels deep are still executing, having sampled the

done blackboard too "early."

It seems that we could work around this problem, however, by checking for completion at multiple levels (*i.e.*, at the previous choice point and its ancestors). The following implementation of the cryptarithmetic solver performs these checks. It does so by passing descendant (recursive) calls to **solve_crypt** a *list* of blackboards instead of a single blackboard. Recursive invocations of **solve_crypt** then treat the head of the list as the blackboard recognizing completion of the immediate parent, while blackboards further down the list represent completion at higher levels of the search. Figure 2.10 shows a **done** list that might be seen by an invocation of **solve_crypt** four levels deep in the search; completion at three levels deep has been noted, while none of the other levels has yet recognized completion of the search. The **solve_crypt** procedure then uses **any_done?** to check all blackboards on the list for completion, fulfilling our goal of looking for completion in all ancestors of the current search node.

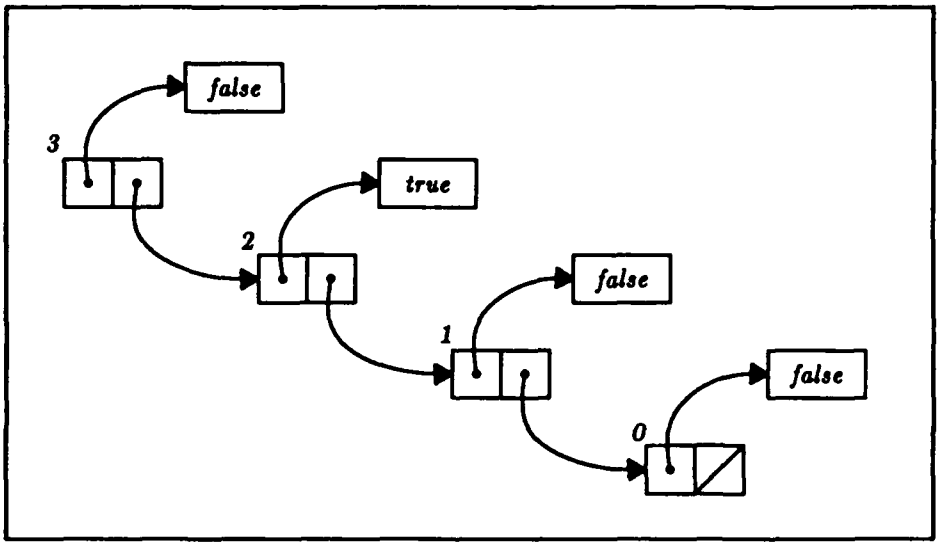


Figure 2.10: List Structure Representing Multi-Level Completion Blackboards

The code to implement this approach in ID is:

```
% Hybridized cryptarithmetic solver, using deep communication.
def crypt a1 a2 a3 =
    { cell = (blackboard false) : nil;
      r1 = reverse a1;
      r2 = reverse a2;
      r3 = reverse a3
      in
         solve_crypt r1 r2 r3 (merge r1 r2 r3) cell nil };

% New version of solve_crypt without the if serialization
% but with deep cutoff controls using blackboard.
def solve_crypt l1 l2 result letters done bindings =
    if any_done? done
      then nil
      else if not (check_solve l1 l2 result bindings)
             then nil
             else if letters == nil
                    then bindings
                    else { letter : rest = letters;
                           _, digits_used = unzip2 bindings;
                           % Generate new list of completion blackboards.
                           cells = (blackboard false) : done;
                           def try nil = nil
                           |   try values =
                               { v : vt = values;
                                 answer = solve_crypt l1 l2 result rest cells
                                                      ((letter, v) : bindings);
                                 another_answer = try vt
                                 in
                                    % Notify of completion if appropriate.
                                    if answer ≠ nil
                                      then { (hd cells)[0] := true in answer }
                                      else another_answer }
                           in
                              try (difference (0 to 9) digits_used) };

% Check if any ancestor is finished.
def any_done? nil = nil
|   any_done? first : rest =
    if first[0]
      then true
      else any_done? rest;
```

We now have a cryptarithmetic solver with deep completion-checking; a solution at any level will eventually be noticed by all descendant search levels, which will then cut off usage of

system resources. However, we have now increased the complexity of the code with quite a few extraneous checks, and have caused a per-search-level overhead to check for completion in all ancestors. This seems "backwards;" if we could instead perform some sort of *forward* communication of completion only when a solution is *found* in a subtree, we could cut down this overhead.

Since each level of the search described by **solve_crypt** knows precisely how many descendants it will have, we could use an alternate data structure. In this approach, each recursive level of **solve_crypt** constructs a vector; element zero of that vector, as before, records nondeterministically whether that level of the solver has completed. The rest of the elements, however, point to the vectors constructed by *descendant* invocations of **solve_crypt** (by specifically storing into various indices of the parent's vector). The value *nil* is stored at the leaves of the search. Figure 2.11 shows the state of such a search through the use of these vectors. The root node *0* has not completed the search (therefore shows *false* in index zero); it has three descendants (labeled *1a* through *1c*). Some of the level-one searches have terminated (signified by the null entries in their vectors), while some of the level-one search is still expanding (signified by the empty cells, to be filled in by a descendant recursive call to **solve_crypt**). Node *1b* has found a solution, and has communicated that fact to node *2b* by its forward pointer.

The following ID code implements the cryptarithmetic solver with "forward communication." Each level of the solver keeps references to all search subtrees so that it can communicate completion to them, using the data structure discussed.

```
% Hybridized cryptarithmetic solver, using forward communication.
def crypt a1 a2 a3 =
    { cell = blackboard false;
    cells = i_array (0, 1);
    cells[0] = cell;
    r1 = reverse a1;
    r2 = reverse a2;
    r3 = reverse a3
    in
       solve_crypt r1 r2 r3 (merge r1 r2 r3) cells 1 nil };
```

%%% This code is continued on page 50.

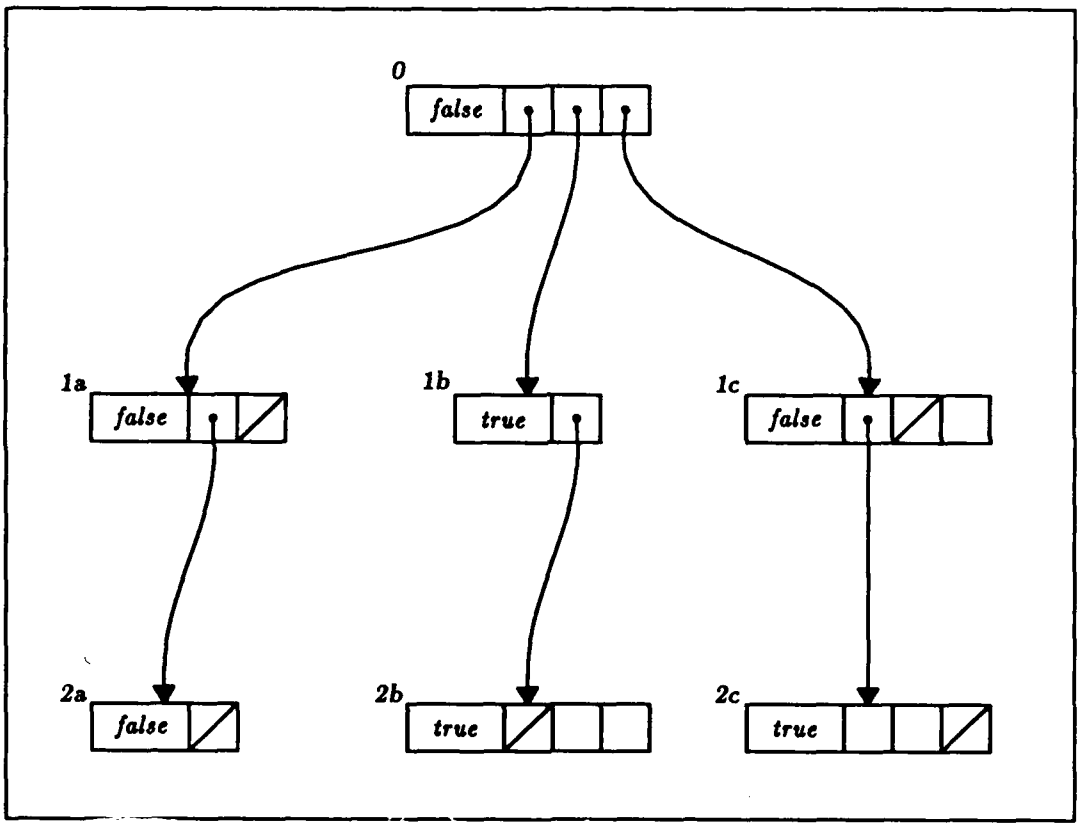


Figure 2.11: Vector Structure Representing Forward-Communication Blackboards

%%% This code is continued from page 48.

```
% New version of solve_crypt without the if serialization
% but with deep forward cutoff controls using blackboard.
def solve_crypt l1 l2 result letters done index bindings =
    if done[0][0]
      then { done[index] = nil % Leaf of termination tree.
             in nil }
      else if not (check_solve l1 l2 result bindings)
             then { done[index] = nil in nil }
             else if letters == nil
                    then { done[index] = nil in bindings }
                    else { letter : rest = letters;
                           _, digits_used = unzip2 bindings;
                           cell = blackboard false;
                           possibilities = (difference (0 to 9) digits_used);
                           cells = i_array (0, (length possibilities));
                           cells[0] = cell;
                           done[index] = cells;
                           def try nil = nil
                           |   try i values =
                               { v : vt = values;
                                 answer = solve_crypt l1 l2 result rest cells i
                                                          ((letter, v) : bindings);
                                 another_answer = try (i + 1) vt
                                 in
                                    if answer ≠ nil
                                      then { call termination cells in answer }
                                      else another_answer }
                           in
                              try 1 possibilities };

% Communication termination "forward." The 0th element of the
% "cell" holds the completion indicator for the ancestor level,
% while the balance of the cell holds pointers to descendant levels.
def termination cells =
    if cells == nil
      then nil
      else { cells[0][0] := true;
             _, descendants = bounds cells;
             { for i ← 1 to descendants do
                   call termination cells[i] }};
```

We have followed a very structured approach to generating a powerful hybrid solution for

the cryparithmetic solver. The steps were:

1. Generate the simple sequential solution.
2. Add parallelism by removing the successor constraint in the search. Though our critical path shortened significantly, we increased parallelism to a ludicrous extent. And we ignored knowledge about the search progress, a dangerous approach.
3. Begin to control that parallelism (in a one-deep fashion) by using the **blackboard** construct to communicate completion to a child invocation of the solver. This helps, but has a timing problem—the child may start and do its completion check just before the parent signals completion.
4. Checking for completion at all ancestor levels fixes this problem. But it introduces too much overhead in deep searches.
5. Using forward communication of search knowledge alleviated that problem, with an attendant cost of program complexity.

Yet the realization of this simple hybrid approach is beginning to be incomprehensible. Worse, we haven't yet hit the bottom of the well; the solution above doesn't do anything to stem the usage of resources in functions other than **solve_crypt**. In the general case we might want to check for search completion (and perhaps notify of search completion) in many, many functions; should all ID procedures be recompiled for usage with search to check for completion? That approach seems excessive. Nevertheless, it seems not to insure that the propagating termination information will "catch up" with the continued unfolding of the execution tree, and therefore might not even work.

We should be able to design a single construct for speculative search of the type we're interested in that would encapsulate and hide the details of nondeterminism and control from the programmer, while still allowing (and *guaranteeing*) that control.

2.5 General Search Speculation

Efficient general constructs for controlled speculation, with less work required of the programmer, are the goal of this report. In this section we introduce such a construct, and

follow with how this organization may be realized in the ID language with lower-level language forms.

A general tool for speculative search would be nice in order to avoid the explicit parallel nature of the hybrid tree-search code. By introducing some more syntax into the language, we can provide the power of speculative search with controlled parallelism with two important features:

- Abstraction: a single syntax can control all speculative parallelism, and contain the control features we decide to implement.
- Encapsulation: all non-functional operators (such as :=) can be hidden inside this abstraction, leaving the user to write purely functional codes (increasing the probability of correct code and the ease of debugging).

The **speculate** form is defined simply in a manner analogous to list mapping functions found in ID or LISP. The following form

---

speculate f list

---

is defined (operationally) to call the function **f** on each list element of the argument **list**, returning one of the results. This is quite similar to the *EITHER* construct of Baker and Hewitt[15]. Given the tone of this report, there is of course an operational implication that all of the applications of **f** to elements of **list** will begin in parallel, with unfinished branches being "terminated" once another branch finishes.

Specifically, in a manner similar to the description of **blackboard** above, the **speculate** form presents the following operational interface:

---

speculate ((+) 1) (1 : 3 : 5 : *nil*)

$\Longrightarrow$ ((+) 1 1) *or* ((+) 1 3) *or* ((+) 1 5)

$\Longrightarrow$ 2 *or* 4 *or* 6

This operational definition, however, sidesteps many implications of **speculate** on the ID language--what, for instance, does **speculate** return if its **list** argument is an empty list? What is the result of branches taken inside the **speculate** form that are "terminated?" To

solve this problem, we introduce a new ID basic type *NOVALUE*, with a single exemplar of that type represented by the symbol ℵ.[†] ℵ may be the result of any operation inside an execution tree which is undergoing termination.

We might then take an approach akin to the explicit representation of run-time error values in the denotational semantics literature for specifying the semantics of expressions which might undergo termination. For example, in Gordon's TINY language[37] the semantic clauses of the language all must explicitly represent the possibility of run-time error output. The denotational behavioral description (*i.e.*, the mapping $\mathcal{E}$) of the TINY addition expression, for example, becomes[‡]

$$\mathcal{E}[\![E_1 + E_2]\!] = (v_1 = \mathcal{E}[\![E_1]\!]) \rightarrow$$
$$((v_2 = \mathcal{E}[\![E_2]\!]) \rightarrow$$
$$(\text{isNum } v_1 \text{ and isNum } v_2 \rightarrow$$
$$v_1 + v_2, \text{error}), \text{error}), \text{error}$$

We could take the same approach in specifying **speculate**'s static semantics (type). In particular, the static semantics of **speculate** could be specified in this manner as

---

typeof speculate $= (*_0 \rightarrow *_1) \rightarrow (\text{list } *_0) \rightarrow *_1 + \mathit{NOVALUE}$;

---

where the + operator specifies the disjoint sum of types. We must also, however, describe the impact of ℵ on the rest of the ID language. In particular, we intend to specify that dataflow operators are in general *strict* in the *NOVALUE* type[§] (strictness of dataflow operators will be discussed in detail in Section 4.1). This would therefore imply that any and all ID language constructs might have the result ℵ; this would have to be reflected in the type signatures of every operator in the language, as in

[†]This symbol, the Hebrew character "aleph," should be pronounced *no value* in this report.

[‡]For the purposes of this discussion, we ignore the issues of state transformation in the TINY language.

[§]I.e., a dataflow operator that has a ℵ input will have a ℵ result. There will be numerous counterexamples, however, including structure storage, explicit tests for ℵ, *etc.*

typeof + = N + *NOVALUE* → N + *NOVALUE* → N + *NOVALUE*
typeof − = N + *NOVALUE* → N + *NOVALUE* → N + *NOVALUE*
typeof * = N + *NOVALUE* → N + *NOVALUE* → N + *NOVALUE*
typeof / = N + *NOVALUE* → N + *NOVALUE* → N + *NOVALUE*
typeof == = N + SYM + *NOVALUE* → N + SYM + *NOVALUE*
→ B + *NOVALUE*
⋮

This cumbersome construction would be extended to generating a disjoint sum of the *NOVALUE* type with every other type in the language. However, this approach is really useless. The only reason we would need any type specification to explicitly represent the *NOVALUE* type would be to infer properties of programs which might take or return ℵ. But by definition *all* ID programs might take or return ℵ; if in fact we could infer that a particular program would *definitely* return ℵ then speculation over a branch of code including that program would be useless!

Therefore, for the purposes of specifying static semantics, we will ignore the *NOVALUE* type. Where a type checker for our improved language specifically infers type *NOVALUE*, it would treat the result as being of unspecified value (unifying with any other type), allowing the dynamic behavior to be ignored by the static semantics. This approach to the treatment of ℵ is quite similar to the treatment of statically recognized dynamic errors of omission discussed by Nikhil.[56]

We will continue to use the symbol ℵ, however, to help specify the *dynamic* semantics of various constructs, including **speculate**. Some properties of **speculate** therefore include the following, where the symbol ⇒ represents evaluation:

% Static semantics:
typeof speculate = $(*_0 \rightarrow *_1) \rightarrow (\text{list } *_0) \rightarrow *_1$;

% Dynamic semantics:
typeof speculate = $(*_0 \times *_1) \times (\text{list } *_0) \Rightarrow *_1 + \{ \aleph \}$;
and:
1. ∀f, **speculate** f *nil* ⇒ ℵ;
2. **if** ∀x ∈ l, f x ⇒ ℵ **then speculate** f l ⇒ ℵ
3. **if** ∃x ∈ l such that f x ↛ ℵ **then speculate** f l ↛ ℵ

We will explore the affect of ℵ on dataflow operators and compilation of the ID language in Chapter 4. Meanwhile the reader may simply assume that ℵ objects propagate through a dataflow execution graph in a well-behaved manner, as has been implied in various papers in the dataflow literature.[19]†

For the balance of this chapter we will explore the use of **speculate** and related forms.

2.6 Using Speculation

This approach is perfect for solving problems which have two or more known solution algorithms, with neither being guaranteed optimal in time. For example, a symbolic integration system might define a general integration function as[15]

```
def integrate expression =
    { def try expression f =
          f expression
      in
         speculate (try expression) (heuristic_integrate : Risch_integrate : nil) };
```

This function would then try two general methods of symbolic integration, returning the result of the one that finishes first.

More generally, the **speculate** construct is a powerful approach to searching a tree in which any one result is acceptable. For example, we can re-write our cryptarithmetic solver's top-level function quite simply now:

†In fact, the simulation of the Tagged-Token Architecture used to produce results for this report simply used the error-propagation mode of the simulation system.

```
% Solve a crypt puzzle recursively, using speculate.
def solve_crypt l1 l2 result letters bindings =
    if not (check_solve l1 l2 result bindings)
      then nil
      else if letters == nil
            then bindings
            else { letter : rest = letters;
                   _, digits_used = unzip2 bindings;
                   def try possibility =
                       solve_crypt l1 l2 result rest ((letter, possibility) : bindings)
                   in
                       speculate try (difference (0 to 9) digits_used) };
```

Assuming that generating and checking for puzzle solution is relatively expensive, the value of the **speculate** form is in executing all generations and tests in parallel, yet terminating all still-running tests once a particular search branch succeeds. Although this latter ability doesn't affect the semantics of the solution (as the hybrid alpha-beta algorithm didn't affect the semantics of minimax tree search), efficiency gains can be made. The multiple calls to **try** are said to be *interspeculative*, as they represent the search of the disjoint subtree at any given point in the search space. Furthermore, we specify that each application of the **speculate function** argument to an element of the **list** argument, and the resulting execution tree, represents a *task*.[†] Thus we can speak of task termination in a well-defined manner. In addition, any task can spawn further *subtasks* by suitably calling **speculate**; we define the termination of a task, for our purposes, to include the termination of all subtasks of that task.

In fact, there is something a bit strange about task termination as it has been presented. We allow the **speculate** form to terminate tasks somehow, but we give the user no tool to indicate to the system that the *current* task is known not to be able to generate a successful solution (*e.g.*, in the **check_solve** procedure of our cryptarithmetic solver). We introduce another form, **terminate**, to allow this indication. We might define the functional semantics of **terminate** as follows:

[†]To some extent we rely on the reader's intuition about what constitutes a task. We will back this intuition with a better description in Chapter 3.

---

typeof terminate = $*_0 \rightarrow *_1$;

---

ignoring the side-effect of **terminate** (that is, the notification of termination). This specification shows a single argument; that argument is actually a *dummy argument*, used simply to trigger the execution of the **terminate** function.† We will generally specify *nil* as the argument to **terminate**, although any argument will do as it is ignored by the program.

The following dynamic semantics of **terminate** would serve us better than the semantics given above:

---

typeof terminate = $*_0 \Rightarrow \aleph$;

---

as it would cause immediate injection of $\aleph$ into the computation at the point of specification. This is just an optimization, however, and either approach is acceptable. Using this new construct, we can re-write some of our cryptarithmetic solver even more simply:

†This is due to an artifactual bug in the syntax of ID which disallows the specification of a *call* to a non-constant (*i.e.*, side-effecting) code-block with no arguments.

```
% Speculating cryptarithmetic solver using self-termination.
def solve_crypt l1 l2 result letters bindings =
    { call check_solve l1 l2 result bindings in
      if letters == nil
        then bindings
        else { letter : rest = letters;
               _, digits_used = unzip2 bindings;
               def try possibility =
                   solve_crypt l1 l2 result rest ((letter, possibility) : bindings)
               in
                   speculate try (difference (0 to 9) digits_used) }};

% Version of check_solve using self-termination.
def check_solve l1 l2 result bindings =
    { def check nil l2 result carry = carry == 0
      |   check l1 l2 result carry =
          { v1 = lookup (hd l1) bindings;
            v2 = lookup (hd l2) bindings;
            vr = lookup (hd result) bindings
            in
              if (v1 < 0) or (v2 < 0) or (vr < 0)
                then true
                else { total = v1 + v2 + carry
                       in
                         if (mod total 10) ≠ vr
                           then terminate nil
                           else check (tl l1) (tl l2) (tl result)
                                      (floor (total/10)) }}
      in
        check l1 l2 result 0 };
```

The **terminate** form is what we term a *post-speculative* control construct, one of the two phases of speculation control. The **terminate** structure is post-speculative in that it checks execution of a task which is *known* not to be capable of producing a correct response, and thus may be terminated (in the sense that no new resources should be allotted the task). The **priority** form, introduced and explained in Chapter 3, is a *pre*-speculative task priority control device. In the *pre-speculative* phase, before it is known which of a group of interspeculative tasks will compute the correct result, there is the problem of constraining the resource usage of each interspeculative task against each other interspeculative task. We shall see some example usage of the **priority** form in Chapter 5.

2.7 Another Example

As another example of the use of speculation, the following ID code implements a solution for the n-queens problem. The aim of this well-known problem is to place n queens on an n by n chess board such that no queen is threatening another (under the standard rules of chess). Typically solved by a backtracking approach, the solution below instead uses the **speculate** form we have introduced into ID.

The solution is straight-forward. The toplevel function **queens** takes as input the parameter n, the size of the problem. It then calls the recursive function **place_queen** for each possible placement of a queen on the first row of the board. **place_queen**, after determining whether the problem has been solved or cannot be solved, then speculates over each of the possible placements of a queen in the *next* row. In this way, we either come to a "dead" branch (wherein not all of the queens have been placed, but already we have queens threatening each other) or to the end of the game, with all queens placed in non-threatening positions. The **threatened?**, **threatened_diagonally?** and **member?** are auxiliary functions which check for positions threatened under the standard rules of chess.

Figure 2.12 shows a typical search tree for the 4-queens problem.

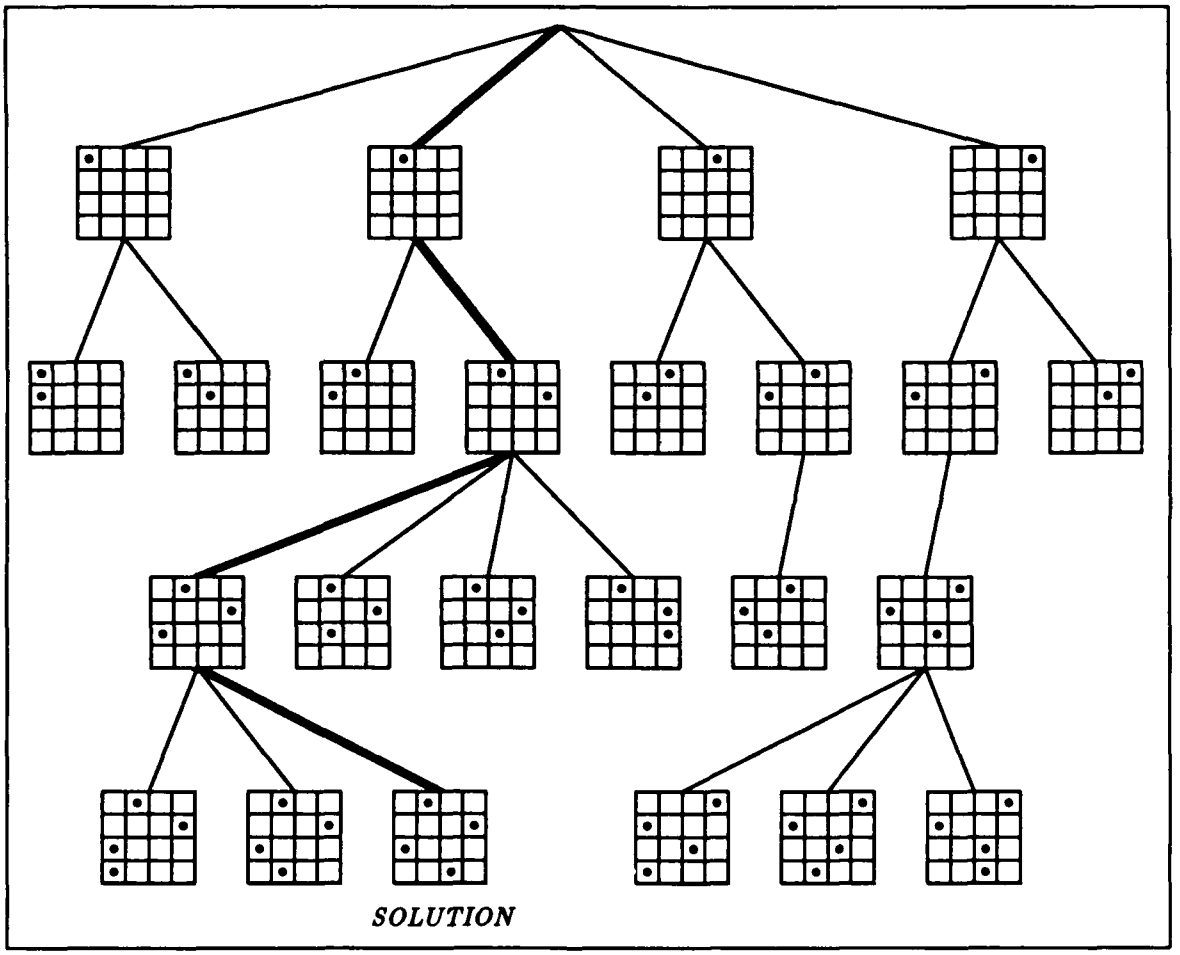


Figure 2.12: Typical Search Tree for the Four-Queens Problem.

```
% Call to solve four-queens problem.
solution = queens 4;

% Solve the n-queens problem.
def queens n =
    speculate (place_queen 1 n nil) 1 to n;

% Check if done; if not, speculate on placement of next queen.
def place_queen row size placements column =
    if row > size
      then placements
      else if threatened? row column size placements
             then terminate nil
             else speculate
                    (place_queen (row + 1) size (row,column):placements)
                    (1 to size);

% Check if a placement is threatened from previous placements.
def threatened? row column size placements =
    { _, columns = unzip placements
      in
        member? (==) column columns
        or threatened_diagonally? row column 1 size placements
        or threatened_diagonally? row column -1 size placements }

% Check if a placement is threatened from diagonal placements.
% We only have to search previous rows.
def threatened_diagonally? row column dy placements =
    { def test (r1, c1) (r2, c2) =
              r1 == r2 and c1 == c2;
      answer = false;
      x = row - 1;
      y = column + dy
      in
        { while x > 0 and y > 0 and y ≤ size
            next answer = answer or member? test (row, y) placements;
            next x = x - 1;
            next y = y + dy
          finally answer }};

% Test for list membership given a particular element comparison.
def member? test number list = fold (or) (map_list (test number) list);
```

In this chapter, we have seen the kind of speculative parallelism we wish to exploit with ID programs, and the type of control over that parallelism that we expect. We built two

different approaches to execution control; the first (blackboard) approach was difficult to use and didn't solve all of our problems. The latter approach (*i.e.*, **speculate**) gives the control we want by definition, but clearly requires support in the language and the run-time system.

In the following chapter, we shall see how these speculation control forms can be implemented within the ID language. Changes in the underlying dataflow architecture to support these constructs efficiently will also be explored.

Nondeterminism means never having to say you're wrong.
— SEAN PHILIP ENGELSON

Chapter 3

Implementation within the Language

In this chapter we explore the realization of the speculation control devices suggested by Chapter 2. We will begin by implementing these constructs within the language, and then investigate the architectural support necessary for efficiency in Chapter 4.

3.1 Is Speculation Trivial?

Initially, it seems that the **speculate** form could be realized by a simple definition:

```
% First stab at definition of speculate.
def speculate function list =
    function (choose list);

def choose list =
    { element_index = random (length list)
      in
        nth element_index list };
```

where the function application **random n** function returns (perhaps) uniformly distributed random numbers in $[0, n)$. This trivial **speculate** function would simply choose a random element of the input list, and call **function** with that element. Clearly this definition returns the **function** applied to an element of the list.

However, this definition fails both the semantic definition of **speculate** as well as the

operational requirements of our new form. No provision is made for the behavior evidenced when **function** applied to the list element does not generate a value (through error or an explicit ℵ); our special **speculate** form is supposed to return the result of applying **function** to a list element *which generates a value.* Operationally, the definition above might also choose the "wrong" element to operate on, in the sense that another element might have proved to return an answer sooner.

Another stab at the problem, which successfully fulfills the semantic requirements by using the mechanisms of I-structure storage, might be written as in the following:[†]

```
% Explosive definition of speculate.
def speculate function list =
    { result = i_array (0, 0);
      def task element =
          { answer = function element
            result[0] = if not (novalue? answer)
                        then answer
                        else result[0] };
      map_list task list
      in
         result[0] };
```

By using I-structures, and depending on the synchronization feature of I-structures (*i.e.*, that an I-structure cell may only be written once), this proposed solution will correctly return the result first returned by the applications of **function** on the elements of **list**. All other applications of **function** will fail in their attempts to write the **result** cell, getting run-time errors which will eventually cause termination.

This completely ignores the question of how the I-structure error signalled by such a code is treated. Does all computation terminate? Does the machine halt? There is not current literature on error handling in Id, although "error propagation" is glibly spoken of in various papers.[19][‡]

Worse, however, the above definition does not fulfill the *operational* goal. Once *any* of

[†]We ask the reader to give an intuitive meaning to the "novalue?" predicate; we will see the details of this procedure in Chapter 4 and Appendx A.

[‡]Plouffe's work[64] contains a more complete discussion of error handling and propagation in functional languages. This report, however, will define a propagation semantics for ℵ, which may be thought of as a subclass of the class of errors.

the interspeculative tasks returns a result, we wish all other tasks to *stop consuming resources.* This solution does *not* gain that effect, as post-speculative (and therefore *known to be garbage* tasks) will continue spending system resources until they attempt to return a result. Worse, because of the generally non-strict argument-passing semantics of ID [74], a post-speculative task could continue to use resources even after generating an error while returning a result.

How can we specify this operational goal in the language in a consistent manner, useful for controlling execution in general, and fitting in to the ID language as a whole? With the addition of certain inherently nondeterministic constructs and the augmentation of various system-level structures, the ID language is itself a suitable medium for implementing the **speculate** routine and its approach to parallelism control.

In any language under any execution paradigm, application of a procedure (function calling) requires some set of system resources. The work of the operating system in supporting a function call might be as trivial as allocating an activation frame (*e.g.*, stack frame), or it might be as complex as mapping the function "name" to an address (dynamic linking), checking access, copying arguments, *etc.* Regardless, some entry into the system resource management system is necessary. This continues to be true in the ID language under dataflow execution. Obviously, this resource control issue must be central to the prioritization and termination of tasks as well.

For the next several sections of this chapter we will digress, discussing various ID language features considered necessary for our purposes. We will explore argument strictness issues, resource management in the ID language, and dynamic enclosing constructs which will give form to the intuitive "task" idea we have used previously. In Section 3.6 we will pull together these lower-level constructs into an ID-language implementation of **speculate** that meets the semantic and operational goals we require.

3.2 Strictness-Enforcing Constructs

Procedures in the ID language are non-strict in their arguments,[†] in the sense that function bodies may begin executing—and even return results—before all arguments have been

[†]To be more exact, ID procedures are *lenient* in their arguments. For a more full treatment of this topic, see Traub.[74] We will simply use the term *non-strict* in this report.

supplied to the function.[58] For example, in the following code

```
def f x y = x;

f 3 ⊥
```

the call to the function **f** will return **3**, regardless of the fact that the second argument to **f** was ill-defined. Though this feature has major repercussions in the architecture of the compiler and implementation for the ID language,[74] it is quite powerful and allows simple, declarative specification of difficult concepts. In fact, arguments to ID procedures may even depend on the results of the procedure call. In particular, a circular list of sevens may be constructed in ID via the following expression:

```
{ result = cons 7 result
  in
     result }
```

This fragment of proper ID code clearly constructs and returns a **cons** (list element) containing **7** in its head, and itself in its tail. This behavior depends on the non-strict argument passing behavior of **cons** and every other ID function, whether system- or user-defined.

However, this feature has drawbacks. In particular, once the language contains nondeterministic features (which we have already seen), control-based time-ordering of certain code fragments becomes important. The partial-order of execution scheduling represented by the dataflow graph is gleaned *only* from data dependency information; the *control* information needed for careful use of nondeterminate constructs cannot be automatically computed by the ID compiler.†

Hence we introduce into the language a construct named *gate* to allow explicit language-based sequentialization of function calls and other operations. The code fragment

gate value trigger

†This is not true in some specialized cases, such as input/output ordering. See the author's I/O serialization work[66] for more information.

is defined to return its first argument (**value**) as soon as the second argument (**trigger**) has arrived. We will see the use of this functional, safe construct in many implementation details below.

The implementation of this construct relies on the dataflow execution model underlying the ID language. Although the ID language implements non-strict function argument passing, the underlying execution paradigm for dataflow graphs is itself *strict*. That is, no dataflow operator will "fire" (execute) until all of its arguments are available. Thus the ID compiler can simply use a gating instruction which passes a value through untouched upon receipt of a second argument. This instruction is represented in dataflow graphs by the symbol shown in Figure 3.1, and directly implements the **gate** functionality discussed above.

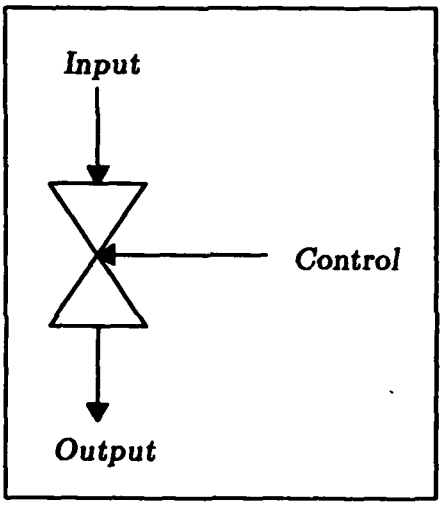


Figure 3.1: The Gate (Strictness Imposition) Operator

3.3 Managers

The central important feature of ID that we need to exploit in order to control speculation is the concept of a *manager*. Though managers have not been fully explored in the dataflow or functional language literature,[10, 5, 13, 23] they have been explored in other media[16] and actually implemented on a trial basis under various dataflow systems.[61, 55, 57] Since the **speculate** form is intimately dependent on the semantics of managers, we include here a discussion of how managers operate, and the effect of managers on system-code implementation. In a later section (Section 4.2) we will explore the implementation of

these nondeterministic structures. We believe that the nondeterminism encapsulated by managers is required for reasonable resource allocation control; in fact, it has been said that determinate computation is not adequate in general for resource allocation.[22]

Managers provide the highest-level interface within the ID language for serializing access to shared resources.[13, 5]. These shared resources may be memory space (or portions of memory, such as *databases*), input/output devices, registers (or other function context information), and so forth. Traditionally, access to these resources is controlled by operating system code, and interfaced to user code via "trap vectors," "software interrupts," "dynamic link names" or other globally-named means. The operating system software must nondeterministically serialize access to these generally non-reentrant codes via some means (*i.e.*, critical section control).

In ID, the **manager** construct provides a methodology for implementing shared resource controllers with two powerful features:

- Managers are created and used in a manner similar to function creation and calling. This makes the writing of managers more simple, which decreases programming errors in them.
- User-written manager code is automatically contained within whatever critical section control is necessary; users need not serialize manager requests (input). In particular, this implies that manager codes can be *deterministic*, leaving the complexities of nondeterministic coding to the lower-level system.

Figure 3.2 shows the general schema of a manager code. All calls to a manager are serialized via some lower-level feature (which will be discussed later). Each request is then presented to the code body of the manager, which may be completely deterministic. Any state necessary for the implementation of the manager is automatically circulated as in loop schemas.[73]

Once the manager body has completed computing the "answer" for a particular request, this result is combined with the tag specifying the caller and sent on to the caller. Thus, to the calling function requesting resources the manager appears as a standard ID function. Yet this simple structure allows fully general nondeterministic resource access with state in the controlling code.

This encapsulation of the nondeterministic behavior of resource controllers is the primary feature of the manager scheme. Nearly as important, however, is the fact that the manager code body, itself deterministic and adhering to the standard ID rules, may be executed

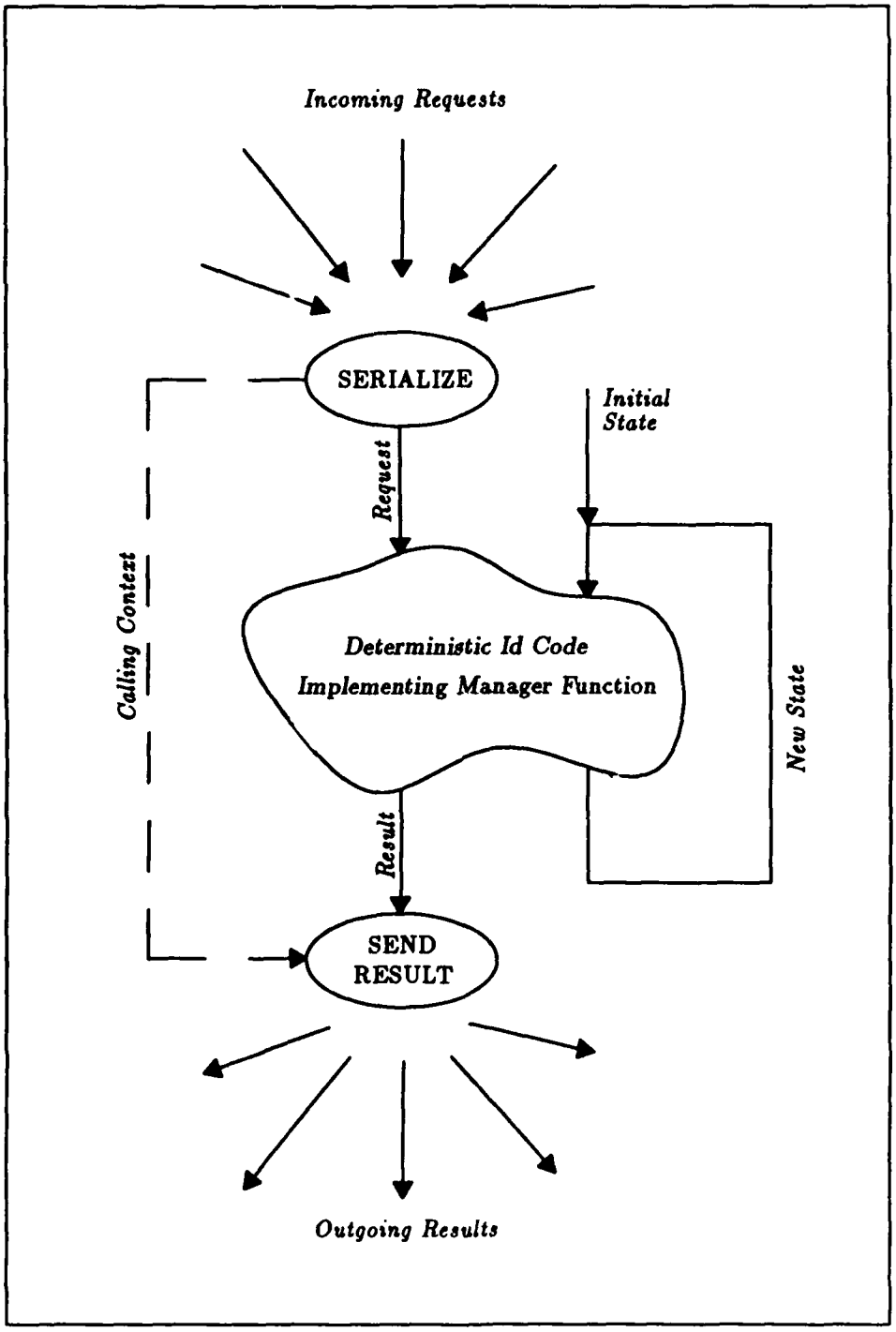


Figure 3.2: General Manager Schema Containing Deterministic Code

with fine-grain parallelism like any other ID code. Thus ID with managers allows parallel execution in managers themselves.

The syntax for manager creation and usage is simple and general. The following code implements the usual bank account example:

```
% Account manager code body. Requests are 'deposit and 'withdraw.
% Returns (true, new balance) if the request is allowed.
% Returns (false, new balance) if the request is not allowed.
def account balance request amount =
    if request == 'deposit
      then true, balance + amount
      else if balance < amount
             then false, balance
             else true, balance - amount;

% Create a bank account with a particular balance and return it.
def new_account initial_balance =
    manager account initial_balance;

% Deposit some amount to an account.
def deposit amount account =
    use account 'deposit amount;

% Attempt to withdraw some amount from an account.
def withdraw amount account =
    if use account 'withdraw amount
      then "Withdrawal succeeded."
      else "Withdrawal failed.";
```

In this example, the function **account** is a deterministic (in fact, functional) code that implements the core functionality of a bank account. When presented with a past state (the **balance** of the account) and a new request to **'deposit** into the account, it increments the account **balance** by the value of the **amount** argument. When presented with a **'withdraw** request, it either performs the subtraction or not, depending on the current balance and the requested withdrawal amount. In any case, this function returns a boolean informing the caller whether the request succeeded, and a new value for the state of the manager.

Thus, new bank accounts are created by the **new_account** function by simply calling **manager** on the bank account function **account**, specifying an initial state (*i.e.*, initial balance).

The **manager** call expands into the nondeterministic schema outlined in Figure 3.2. The two functions **deposit** and **withdraw** then make calls to the manager via the **use** schema, which simply presents the requests to the manager's serializer. The implementation of this serialization structure will be examined in detail in Section 4.4.

This high-level example, though it shows the power of manager-based nondeterministic programming, glosses over the lower-level usage of managers that underlies all ID programming.

3.4 Managers for Basic Language Functions

Since resource access is central to any computer system, the manager scheme is central to the execution of ID codes on dataflow machines. In particular, requests to the system for memory allocation and for code block execution (*i.e.*, function calling) are handled by resource managers which may themselves be written in ID.

In particular, the abstract dataflow graph for implementing the ID function call **f a** (*i.e.*, the application of the function **f** to the argument **a**) involves a call to a manager that implements function call context and argument storage, as is outlined in Figure 3.3.

In the figure, we see two uses of the **use** abstract instruction for calling managers. The first allocates context and argument space for the named function (perhaps loading and linking the function dynamically first); the second deallocates that space for other use. In this simple single-argument case, the result of the first **use** instruction directs two instructions for sending return address information as well as the argument to the function. On the completion of execution of the function, the result is passed to the caller, and is used to start the second call to the application manager to deallocate the function's context and argument space.

Calls to allocate memory (*e.g.*, for array storage) are quite similar in structure; a **use** instruction is used to call the manager to allocate or deallocate memory.

3.5 Hierarchical Naming of Managers

This brings us to the problem of *naming* managers. In the abstract graph of Figure 3.3, we glossed over this point by providing a symbol (**'application_manager**) which names the

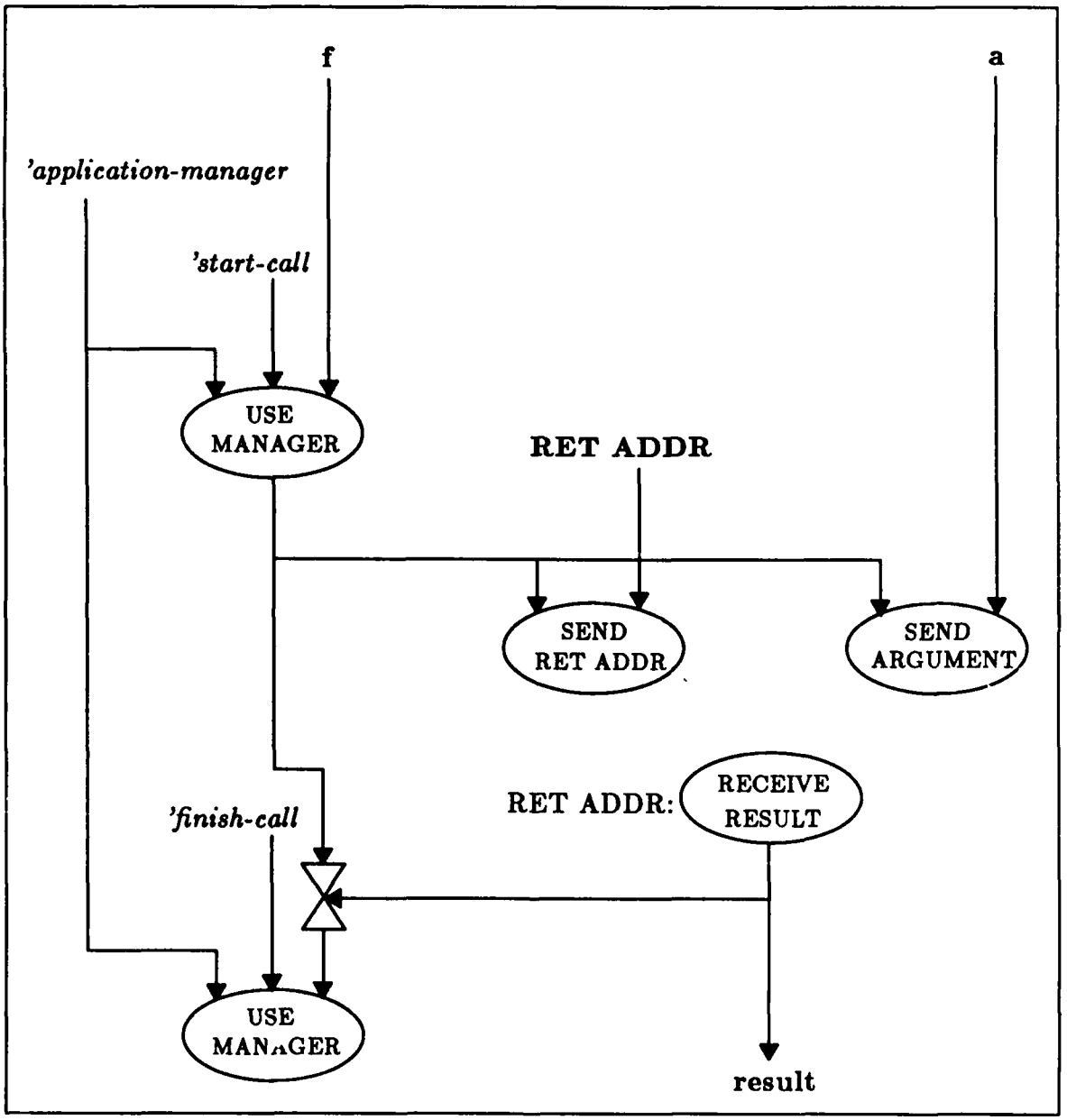


Figure 3.3: Function Calling Abstract Graph

manager to call. At some point, however, this symbol needs to be mapped into an actual serialization structure as outlined in Figure 3.2.

This naming problem has not been resolved in the manager literature.[5, 57] The "obvious" idea, using the same mapping that takes place in order to map function names into their corresponding dataflow graph entry points, cannot work—that is one of the functions of the application manager itself! As was pointed out above, this problem of recursion is generally solved on traditional machines by static naming solutions (*i.e.*, trap or interrupt vectors or global symbolic names). However, static solutions are not powerful enough for our purposes, nor are they in the spirit of the dataflow execution model.

Instead, we use a *dynamic* scoping model to map manager names to manager serialization entrypoints. This is a departure from the name scoping paradigms found elsewhere in the ID language, which include

- *Lexical* scoping, which is used to resolve free variable references inside function definitions, and
- *Global* (or *static*) scoping, which in ID is primarily used to resolve free function name references. In the abstract, global scoping is identical to lexical scoping if we imagine that all top-level function definitions are contained within a single "invisible" all-encompassing lexical scope. Hence the word "global" is used.

However, we find the ability to specify dynamic (call-tree based) scoping for various variables to be a simplifying solution for some problems. Other researchers have also noted that the use of controlled dynamic scoping can in fact simplify programs,[71] and optional dynamic binding of variables is found in production versions of various languages, including COMMON LISP.[70]

In particular, in our formulation of managers, manager name scoping is dynamic. Once a mapping from a manager name to a manager serialization entrypoint is specified, all users of that manager in the same lexical context *and all subsidiary dynamic contexts* see that particular mapping. For example, if we assume that the symbol **application_mgr** specifies the manager that is to be called to perform function application, in the code

```
% Declare application_mgr to be dynamically scoped.
@dynamic application_mgr;

% Define x to simply apply y to its argument, using the current
% (dynamically enclosing) application manager.
def x a = y a;

% Define y to apply z to its argument, using the application manager
% new_manager to gather application resources. Also, dynamically enclose
% the call such that future calls use new_manager also.
def y b =
    { application_mgr = new_manager
      in
        z b; }

% Define z to simply apply w to its argument, using the current
% (dynamically enclosing) application manager.
def z c = w c;
```

the function **x** calls the function **y** using the system-defined application manager; but **y** calls **z** with the manager entrypoint **new_manager** (which is assumed to be globally bound to a manager entrypoint). *In addition*, **z** will call **w** with the new manager specified by **y**, since it is dynamically enclosed by a binding of the **'application_manager** symbol. Figure 3.4 clarifies this point.

Although optional dynamic binding has various uses in a language, this use of dynamic binding to control the mapping of manager names to manager entrypoints will be the only use of dynamic binding we will need in implementing speculation control. In particular, we will use it to "filter" manager requests going to the implicit system function application and storage allocation managers.

In essence, this use of dynamic binding will define the term *task* as used in this report. The concept of a task is difficult to gather in a dataflow execution paradigm, where the number of concurrently executable threads of execution may be in the thousands per processor, and where constant context switching is the norm rather than the exception.† Even in parallel processing specification languages such as MULTILISP [38] or QLISP [35], wherein tasks are not first-class objects in the language, implementation details of tasks become convenient

†The Monsoon dataflow execution engine, for example, is designed to context switch on every *pipeline beat*.[61]

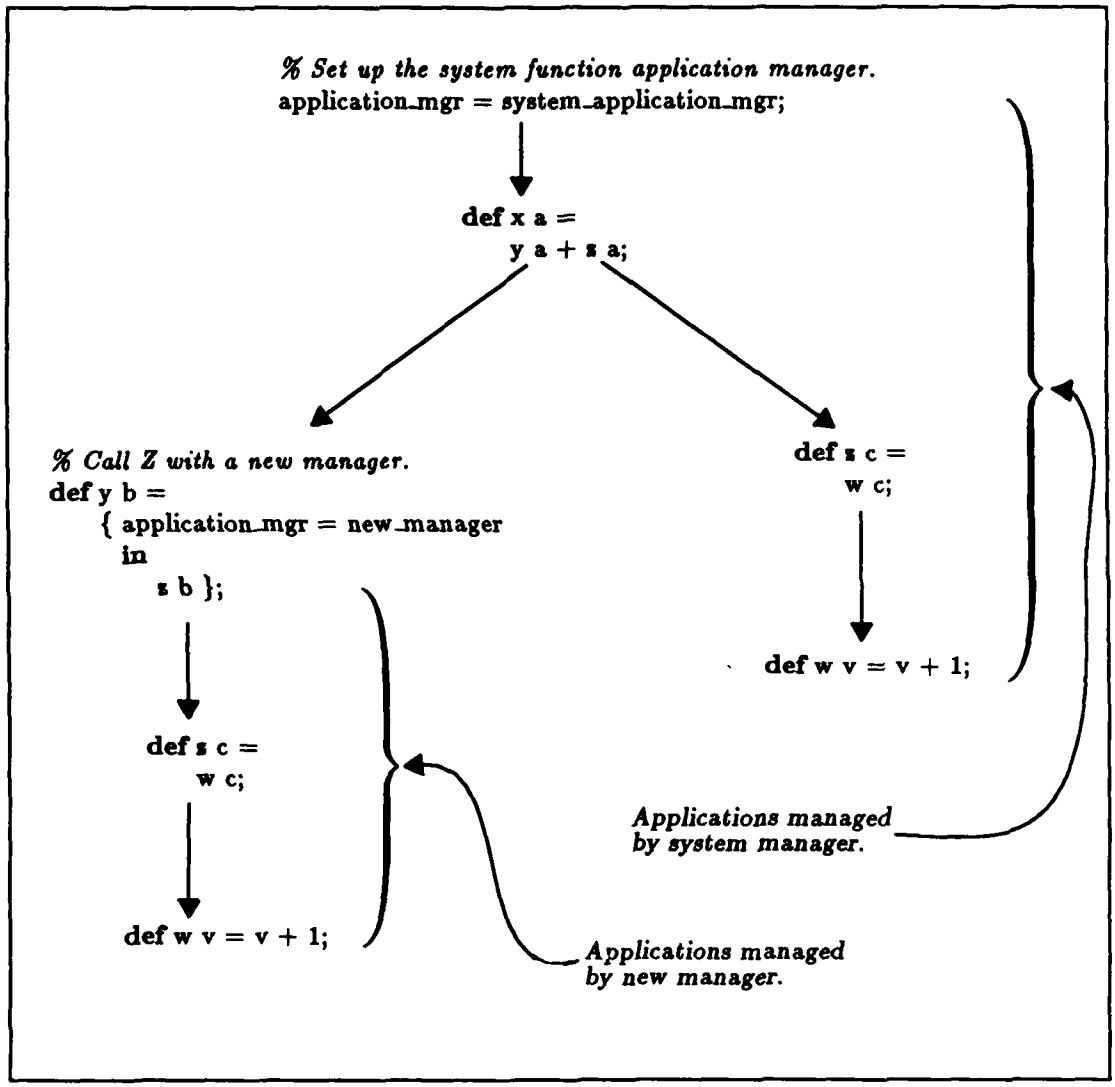


Figure 3.4: Dynamic Binding of Application Management

boundaries for various task-dependent operations such as termination and exception handling. For example, dynamic exceptions across task boundaries in MULTILISP are handled by propagating the exception information through the shared memory point (*future*) of the calling and called task.[40]

For our work, a task is operationally defined to be the set of execution threads dynamically enclosed by a dynamic binding of the application and allocation manager names (which we shall always bind simultaneously). Note that this definition does *not* preclude tasks enclosed by ancestor tasks; we rely on this feature of task nesting to correctly handle nested speculation (since nested speculation, by way of recursion, will be the normal mode of searching a search space).

The central method we shall employ for controlling the growth of a task (relative to its other interspeculative tasks) will be this filtering approach. For example, in the following ID code

```
% Define a new manager code body to filter incoming requests
% based on some predicate allow_request?.
def filter_manager old_manager request =
    if allow_request? request
      then use old_manager request
      else fail_request request;

% Call another function with the same argument, but in a dynamic
% environment in which the filtering manager above is instated
% as the handler for function application management.
def f x =
    { old_application_mgr = application_mgr
      in
        { application_mgr = manager (filter_manager old_application_mgr)
          in
            g x }};
```

the **filter_manager** function will be called to handle function application manager requests. If it decides (based on some predicate **allow_request?**) to allow the request to proceed, it simply passes the request along to the dynamically enclosing application manager (which was passed in when the manager was created by **f**). If the criterion fails, then the manager may disallow the request. It can easily be seen that this basic structure can allow many

kinds of control over execution graph growth.

For the rest of this chapter, we will return from our detour into ID language features, and implement the **speculate** form in the language.

3.6 Implementing Speculate within the Language

When we last visited an implementation of the **speculate** form, in Section 3.1, we saw that ID as previously explored didn't include enough power to match the operational goals of the **speculate** form outlined in Chapter 2. The previous sections outline enough language-level support to fully implement the **speculate** form within the ID language itself. In this section we will explore this realization.

What was missing in our earlier construction was a "daemon" which could observe that a result has been returned from an interspeculative task, and terminate all other interspeculative tasks, restricting their usage of system resources. The following code, which employs nearly all of the new language functionality developed above, correctly implements the basic **speculate** form we need.

In this code, we will use the new primitive *LOCK* to synchronize the returning of results by **speculate**. The basic operations related to *LOCK*, all of which are atomic, include the following:

- **(make_lock value)**: creates a "lock cell," with an initial value of **value**.
- **(lock cell)**: locks **cell**, returning the value currently stored therein.
- **(unlock cell value)**: unlocks **cell**, storing **value** in the cell.

We will see more details on this construct in Section 4.3; until then we ask the reader to rely on an intuitive critical-section semantics for locks.

The procedure **terminate_manager** used by the following and later codes is used to request a manager to return (stop executing); it synchronizes on its second argument, in our case the result of the task.

```
% Definition of speculate with task termination.
def speculate function list =
    { result = i_array (0, 0);
      ntasks = length list;
      termination = make_lock false;
      tasks = i_array (1, ntasks);
      { for i ← 1 to ntasks do tasks[i] = true };

      def task element i =
          { old_apply = application_mgr;
            old_alloc = allocation_mgr;
            in
              { application_mgr = manager speculation_mgr old_apply tasks i;
                allocation_mgr = manager speculation_mgr old_alloc tasks i;
                answer = function element;
                terminate_manager application_mgr answer;
                terminate_manager allocation_mgr answer;
                old_lock_value = lock (gate termination answer);
                result_determined? = not (novalue? answer)
                in
                  if result_determined? and (old_lock_value == false)
                    then { result[0] = answer;
                           remove_other_tasks tasks i ntasks
                           unlock termination true }
                    else unlock termination old_lock_value }};

      { for element ← list & i ← 1 to ntasks
            task element i }
      in
        result[0] };

def speculation_mgr old_mgr control index request =
    if control[index]
      then use old_mgr request
      else terminate request;

defsubst remove_other_tasks tasks winner ntasks =
          { for index ← 1 to ntasks do
                tasks[index] := index == winner }};

def terminate request =
    N;
```

Obviously getting the right result required quite a bit more complexity, although the approach is quite similar to the last attempted solution. The difference can be attributed

completely to the need to terminate execution of tasks which are post-speculative, *e.g.*, which are known to be computing values which are not needed.

A few parts of this solution bear inspection. The general approach is to dynamically bind the implied application and allocation managers within the (thereby defined) interspeculative tasks to filter managers, as outlined in Section 3.5. Thus all requests for function application and storage allocation are handled by the manager constructed inside the **task** function of **speculate**. The code body of this manager (in **speculation_mgr**) then selects the original system manager if the task is still running, or a special termination manager if the task should be destroyed.

These special termination managers are the key to the removal of "dead" post-speculative (but still running) tasks. The difficulty lies entirely in the need to maintain the "well-behaved dataflow graph" concept.[13] Where numerous other paper attempts to define termination constructs for parallel systems have concentrated on "killer tokens" and special associative wait-match "destruction" controls[4, 49] or garbage-collection based dead-tree search schemes[15], we wish to remain within the language in our approach to causing post-speculative tasks to terminate. This will help in maintaining the well-behaved (self-cleaning) dataflow graph invariant, which holds that

- Initially, the program graph is empty (contains no tokens, either waiting to be matched or executing).
- Given exactly one token on each input of each procedure, the procedure will produce exactly one token on each output.
- Once output tokens have appeared on every output of a procedure, the procedure's graph is again empty (contains no tokens). This is the so-called *self-cleaning* property of dataflow graphs.

These properties are central to the way in which tagged-token dataflow architecture memory and process resources are controlled, as well as to the approach to debugging taken by the machine. The importance of maintaining these assumptions is paramount. Therefore we choose a solution to application and allocation control which maintains these invariants.

In particular, the **speculation_mgr** function, called for allocation requests from a "dying" task, does not actually allocate any memory, but instead returns $\aleph$. Similarly requests for contexts receive only a $\aleph$ result, returned by the special function **terminate**. This causes special problems in the way that functions are called in the tagged-token regime. As can be

seen in Figure 3.3, there is an implicit graph continuation (labeled *RECEIVE RESULT*) which receives data from the called function; when the application manager simply returns ℵ, this portion of the graph is never triggered.

The solution, instead, is to rely on slightly different function-calling schemas (and other major language schemas), as well as carefully defined ℵ propagation rules in the operators of the underlying dataflow operator language. We previously stated (in Section 2.5) that dataflow operators are generally strict in ℵ, in the sense that if any input to a dataflow operator is ℵ, the result of the operation will also be ℵ. More detailed analysis of the architecture is necessary to guarantee that graphs remain self-cleaning, however; the necessary architecture clarifications are discussed in Section 4.1.

3.7 Control Forms

Many parallel AI implementation schemes include the need for "killing"[49] or "garbage collection"[15] of tasks which are performing computation that is no longer necessary. Implementing this functionality under the dynamic dataflow paradigm (or *any* paradigm in which the state of computation is often found in the communication network) presents obvious difficulties. We take the same tack as above in providing this feature through the **terminate** special form. This form can be simply expressed using the same managers as previously. We simply add a new manager for handling task termination, as in the following code:

```
% Declare control_mgr to be dynamically bound.
@dynamic control_mgr;

% Definition of speculate with task control manager.
def speculate function list =
    { result = i_array (0, 0);
      ntasks = length list;
      termination = make_lock false;
      tasks = i_array (1, ntasks);
      { for i ← 1 to ntasks do tasks[i] = true };

      def task element i =
          { old_apply = application_mgr;
            old_alloc = allocation_mgr;
            in
              { application_mgr = manager speculation_mgr old_apply tasks i;
                allocation_mgr = manager speculation_mgr old_alloc tasks i;
                control_mgr = manager control_task tasks i;
                answer = function element;
                terminate_manager application_mgr answer;
                terminate_manager allocation_mgr answer;
                terminate_manager control_mgr answer;
                old_lock_value = lock (gate termination answer);
                result_determined? = not (novalue? answer)
                in
                  if result_determined? and (old_lock_value == false)
                    then { result[0] = answer;
                           remove_other_tasks tasks i ntasks
                           unlock termination true }
                    else unlock termination old_lock_value }};

      { for element ← list & i ← 1 to ntasks
            task element i }
      in
        result[0] };
```

%%% This code is continued on page 82.

```
%%% This code is continued from page 81.

def speculation_mgr old_mgr new_mgr control index request =
    if control[index]
      then use old_mgr request
      else terminate request;

defsubst remove_other_tasks tasks winner ntasks =
          { for index ← 1 to ntasks do
                tasks[index] := if index == winner
                                  then 1
                                  else 0 }}

% Manager code body for controlling tasks.
def control_task tasks index request =
    if request == 'terminate_branch
      then tasks[index] := false;

% Now we can write the terminate code itself.
def terminate ignore =
    { use control_mgr 'terminate_branch
      in
        ℵ };
```

As before, the value **tasks[index]** specifies whether task **index** is currently being terminated. (As we saw previously, if this is the case then application and allocation requests for that task are handled specially). Thus we need only add the call to the **control_task** manager, which sets the **tasks[index]** for the current task to *false*. Eventually this will cause termination for that task.

Another control issue strongly related to speculation, particularly in the case of multiple speculative processes in parallel, is prioritization of tasks. Dataflow architectures have traditionally embraced eager, data driven approaches to program execution. When coupled with algorithms that perform little or no speculation, this is generally an unimportant point. However, speculation by definition implies that the results of some tasks to be executed may be unimportant; though an approximation of the probability of usefulness of a task result might be known before task execution, that approximation is often non-zero.

That knowledge of usefulness, however, might provide a perfect control for the overflowing

parallelism of a speculative search. A degenerate example of this is the alpha-beta pruning mechanism presented above. This uses absolute certainty of zero percent usefulness to prune (avoid starting) tasks.

This absolute certainty is hard to find in most systems, however. A. I. researchers have built models of certainty for various search problems[49], and suggested prioritization of tasks based on these approximations of usefulness measures. These measures could be used to control search mechanisms.

However, as was noted above, current dataflow architectures do not have any concept of prioritization of tasks or instructions. Such an approach to the scheduling of enabled dataflow instructions might further cut down instruction overhead in speculative programs without introducing the overhead of lazy evaluation such as that found in current demand-driven execution schemes[42]. We would, however, want to continue to require that dataflow graphs remain self-cleaning.

The **priority** special form, mentioned in Chapter 2, provides a way of controlling pre-speculative priorities of interspeculative tasks. This feature of control is commonly needed for various speculative languages. For example, in Kornfeld's discussion of cryptarithmetic solving programs,[50] it becomes clear that much pre-speculative knowledge is known about the relative merits of various subtrees of any search tree node. In Section 2.4 we dealt with cryptarithmetic solvers while ignoring this information; thus all interspeculative tasks would compete on an equal basis for system resources. This is a foolish when more knowledge is present about the domain.

In particular, Kornfeld suggests that relative processing power in cryptarithmetic solvers should be parceled out to interspeculative tasks in portions based on the following equation:

$$\left((10 - n_1)^2 + \ldots + (10 - n_{10})^2\right)^2$$

for each task where n_i is the number of possible digit assignments for the letter i in the task's situation. Higher numbers in this formulation represent higher priorities (*i.e.*, more access to system resources). In addition, the values are taken to signify priority *ratios* rather than orders; in other words, a task with a priority of $2n$ will get twice the resources of a task with priority n. For simplicity, we will retain this scher

Other A.I. systems for solving speculative problems using terminating and prioritizing systems exist, such as the PARADISE chess-playing program.[77] Wilkins points out that this approach is applicable to many domains, such as other game searches, scene-understanding systems, and so forth. Even in the simple example of Section 2.3 (the alpha-beta hybrid search), a priority system could delay the execution of low-likelihood tasks long enough that they may be proven irrelevant.

The implementation of the **priority** control is not quite as simple as above, however. We now require application and allocation management to maintain prioritized lists of requests. We will gloss over the exact points of prioritization here, and concentrate on the communications which allow us to implant prioritization of interspeculative tasks within the confines of dataflow graphs which retain the self-cleaning invariant. With this said, we can present an implementation of **speculate** which allows termination *and* prioritization:

```
% Declare control_mgr to be dynamically bound.
@dynamic control_mgr;

% Definition of speculate with prioritizing task control manager.
def speculate function list =
    { result = i_array (0, 0);
      ntasks = length list;
      termination = make_lock false;
      apply_queue = make_lock nil;
      alloc_queue = make_lock nil;
      tasks = i_array (1, ntasks);
      % Preset all task priorities to one.
      { for i ← 1 to ntasks do tasks[i] = 1 };

      def task element i =
          { old_apply = application_mgr;
            old_alloc = allocation_mgr;
            in
              { application_mgr = manager speculation_mgr apply queue tasks i;
                allocation_mgr = manager speculation_mgr alloc_queue tasks i;
                control_mgr = manager control_task tasks i;
                manage_priority_queue apply_queue old_apply;
                manage_priority_queue alloc_queue old_alloc;
                answer = function element;
                terminate_manager application_mgr answer;
                terminate_manager allocation_mgr answer;
                terminate_manager control_mgr answer;
                old_lock_value = lock (gate termination answer);
                result_determined? = not (novalue? answer)
                in
                  if result_determined? and (old_lock_value == false)
                    then { result[0] = answer;
                           remove_other_tasks tasks i ntasks
                           unlock termination true }
                    else unlock termination old_lock_value }};

      { for element ← list & i ← 1 to ntasks
            task element i }
      in
        result[0] };

%%% This code is continued on page 86.
```

```
%%% This code is continued from page 85.

% Manage a queue of requests based on priority. Termination has
% already been handled for all requests already queued.
% We omit particular details of priority queue management.
def manage_priority_queue queue manager =
    { while queue ≠ nil do
            % Choose request based on weight of request & task priority.
            request = Choose request which balances queue.;
            use manager request };

def speculation_mgr queue control index request =
    % Call for termination if priority is now zero.
    % Otherwise push request onto the queue.
    if control[index] == 0
      then terminate request;
      else add_to_queue queue request;

% Now we can write the priority function.
def priority new_priority =
    use control_mgr 'set_branch_priority new_priority;

% The terminate function now just sets zero priority.
def terminate ignore =
    { use control_mgr 'terminate_branch 0
      in
        N };

% Manager code body for controlling tasks.
def control_task tasks index request new_priority =
    tasks[index] := new_priority;
```

3.8 Changes to Compilation

An interest' .g side-effect† of adding nondeterministic structures to the ID language is the impact on compilation strategies, particular optimizations as discussed by Traub.[73] In particular, the ID compiler was originally written under the assumption that it would be compiling code for *functional* semantics. Later instantiations assumed side-effects as encapsulated by I-structures.[12]

†Excuse the technical pun.

The changes needed for the **speculate** form and its control structures, however, support a strictly more expressive language, as proven by Ward's discussion of his *EITHER* extension to the lambda calculus.[76] Nondeterminism alters the assumptions for various compiler optimizations as represented in extant ID compilers. In particular, three areas of ID compilation are affected:

- *Fetch Elimination.* This optimization, which assumes that structure cells have write-once semantics, short-circuits I-structure read operations from structure cells that are written in the same procedure. With the presence of the := operator, however, this optimization no longer preserves the program semantics.
- *Code Hoisting.* This common optimization (in other languages as well as ID) moves "loop constant" code within loops outside loop expressions. The unbounded side effects of our nondeterministic constructs, however, disallows the movement of code containing := outside a loop.
- *Common Subexpression Elimination,* or collapsing sections of repetitive code, presents a problem similar to code hoisting for nondeterministic structures.

Although we do not go into more detail about these problems in this report, it is important to be sensitive to the altered semantics of ID we present when designing and implementing a compiler for the language.

More importantly, however, we have not seen how ID may be compiled into dataflow graphs in a manner such as to continue to insure well-behaved self-cleaning execution, in the presence of ℵ propagation. This topic will be covered in Chapter 4.

The id seeks immediate gratification
and operates on what Freud called the pleasure principle.
When the id is not satisfied, tension is produced,
and the id strives to eliminate this tension
as quickly as possible.
— GERALD DAVISON & JOHN NEALE, *Abnormal Psychology, Second Edition*

Chapter 4

Architectural Support for the Implementation

This chapter comprises a discussion of the architectural features needed by a dataflow machine to support the language semantics presented in Chapter 3. Although these changes were not forseen in earlier work presenting dynamic dataflow architecture,[13] some features have appeared in newer dataflow systems like Monsoon[61] and simulators such as Gita[55] through the work of the author and other members of the dataflow research group at M.I.T.

4.1 Assumptions of Dataflow Operator Strictness in $\aleph$

In Section 2.5 we first discussed requiring some general strictness of dataflow operators in a special value $\aleph$. By this we intended that, in general, individual dataflow actors and aggregate operations should always emit $\aleph$ results when any or all inputs happened to be $\aleph$. As we saw in Chapter 3, we depend on this strictness to correctly execute self-cleaning dataflow graphs which are undergoing termination either by other interspeculative tasks (through completion of a speculative search) or through self-termination (the **terminate** expression). This problem is similar to the error-propagation assumed to exist in tagged-token dataflow execution engines.[9, 19] Some treatment of exact semantics for dynamic dataflow graphs in the presence of error inputs has been undertaken by Plouffe.[64]

Our problem is somewhat simpler, however. Since for the purposes of task termination there is no need to remember the source of an error, or to combine $\aleph$ inputs in any way, we can be somewhat more arbitrary in our strictness definitions. In general, all that we

need to ensure is that dynamic graphs of execution continue to *terminate* correctly (*i.e.*, in a self-cleaning manner). In addition, we perhaps would like to allow some static execution graph cutoff in the presence of ℵ inputs to certain graph structures, such as in function calling or looping.

With some changes to the compilation of various structures, and careful definition of the dynamic semantics of all dataflow operators, it is possible to generate graphs which remain self-cleaning in the presence of ℵ inputs. We ask the reader's indulgence in believing that such graph schemata exist, and refer to Appendix A for a more thorough treatment of this topic.

Besides the changes to compilation schemas and clarification of ℵ propagation, we require other alterations to the dynamic semantics of various portions of the tagged-token model. In particular, we wish to expand the purpose of I-Structure memory in two major categories.

4.2 Extensions to I-Structure Semantics

As was mentioned in Section 1.3, I-Structures fill an important rôle in the ID language as the dynamic memory access medium of the language.[41, 14, 13] These split-phase memory operations allow for out-of-order access to memory locations with dynamic naming, letting the programmer ignore issues of producer-consumer synchronization while the run-time hardware manages the heat. The simple semantics of an I-Structure read cycle also fit into the general dataflow paradigm of scheduling operations upon data availability. An I-Structure read of an "empty" (not yet written) cell becomes a deferred continuation which is restarted when the cell is written; this powerful concept is similar to static graph joins used throughout dataflow execution of the ID language.

In order to put into perspective the changes we outline to the semantics of I-structure storage under dataflow regimes, it is important to take this *synchronization* point of view on the previously extant semantics suggested by various authors. Although generally thought of as memory in the data storage sense, in dataflow execution paradigms I-structure fetch and store may be considered dynamically created communication between tasks. That is, they represent dataflow graph arcs that need not be constructed until execution, as opposed to the static graphs created by the ι compiler. From this point of view, for example, we can

see that the assertion by Agha and others[2] that dataflow graphs exclude dynamic graph reconfiguration, a major feature of Actor semantics, is incorrect; the I-structure data storage model, while it removes ID from the ranks of functional languages, may be considered a method of allowing limited reconfiguration of communications paths between ID procedures. I-structures allow late-binding of synchronization which cannot be resolved in the compiler.

From this point of view, we see two missing features of I-structures. One involves repeated communication through the dynamic pathway created by store and fetch instructions; this is the "rewriteable cell" (or **blackboard**) semantics of Section 2.3. In addition, support for nondeterministic merge (serialization) is missing from these communication paths. These features are encapsulated in the ID **manager** and **use** constructs, which require some forms of rewriteable memory and serialization, as we have seen.

We propose to extend the contract of the I-structure storage controller[41] to include the (potentially dangerous) notions of critical section *locking* and unhindered *rewriting* of I-structure cell contents. Such constructs will of course remove the guarantee of determinacy from the language executed on the machine, but will support styles of programming like the hybrid search program outlined above.

Our proposal is to expand the I-Structure Controller to the two new state machines outlined in Figure 4.1. These functions represent the three uses of dynamically allocated memory under this new regime:

- Dynamic program links for "array storage" and program linkage, as is currently supported by I-Structures[41, 14, 61]. This may be implemented via "presence" bits (as mentioned in the Monsoon discussion) to correctly link data producers and consumers.
- Unrestricted memory write access, or updatea ₃ storage. This may be implemented simply by a bypass of the error-checking of I-Structure writes, to allow refinement protocols such as the one outlined in Section 2.3.
- Resource locking interfaces, to support critical-section controlled access to physical resources. An implementation might use the "presence" bits as resource lock bits; we can thus implement locking primitives much like those proposed by Dijkstra.[31, 32, 65] Some related work in resource locking support has also been reported by Jayaraman and Keller.[46]

We must emphasize that multiple use of structure cells is obviously represented by the controller states outlined in Figure 4.1. This multiple access, analogous to multiple interpretation of memory under traditional architectures (*e.g.*, integer and floating-point) must

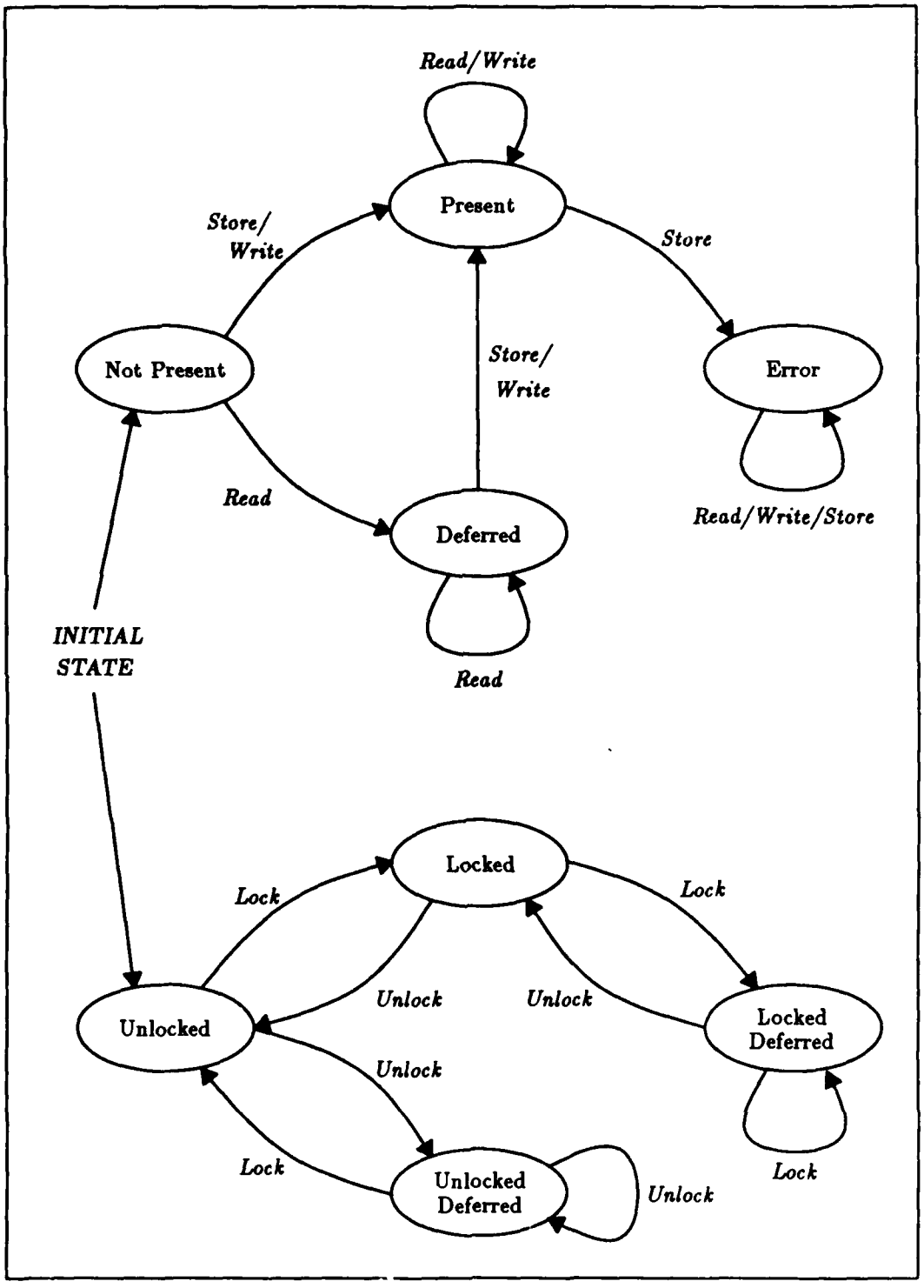


Figure 4.1: Modified I-Structure Controller States

be enforced either by the compiler (in a type-checking fashion) or by programmer convention. We present no architectural mechanism for enforcing correct usage of structure storage.

The simpler of the two modifications to extant I-structure semantics is the addition of the *Write* operation, as can be seen in Figure 4.1. Without adding any new states to the diagram, the I-Structure *Write* request simply allows bypassing of the write-once semantics of I-Structure storage. This allows direct implementation of the **blackboard** semantics of Section 2.3. The := operation is implemented simply as an I-Structure Write operation. We now see that the **blackboard** operator itself is just a general I-structure allocate request, likely with different type, and with use of its slots controlled either by convention or type information.

4.3 Locking Protocol Support in the Architecture

Architectural support for locks is more interesting. The semantics we wish to support include complete critical-section locking as in the work of Dijkstra,[31, 32] but more tightly integrated into the dataflow execution framework.

Typical von Neumann machines, parallel or otherwise, accomplish interlocking with some variant of the *test and set* paradigm;[31] non-interruptible access to particular memory locations on a single-instruction basis is guaranteed for some small set of instructions.[†] These instructions can then atomically test to see that a resource is free to be used, and claim it if so. If not, typically the program needing the resource will "busy wait," or continually attempt to find the memory location free. Unfortunately, this claims processor and memory system time, reduces the performance available for useful work and tends to cause increased latency for claiming access to resources. This is wasteful of CPU resources. Our interlock mechanism obviates the need for busy waiting, depending heavily on the inherently message-based execution model of dataflow.[‡]

A split-phase memory operation not unlike the dataflow I-structure paradigm[41] will allow implementation of locking memory cells that do *not* require busy waiting. As in the I-

[†] Often just a single instruction.

[‡] Such a locking protocol has already been implemented on the Monsoon dataflow testbed,[61] and more work is planned by Steele for future hardware implementations of this protocol.[68]

structure fetch paradigm, lock requests to a cell that is already "locked" will be deferred until the cell is unlocked; unlock requests will wait until a cell is locked. The abstract code below represents the behavior of these lock cells, where the initial state is UNLOCKED. The two instructions that act on these lock cells include

- LOCK (cell): returns only when the cell has been locked, with the value written to the cell when it was allocated or last unlocked.
- UNLOCK (cell, value): unlocks the cell specified, writing the given value into the cell.

The precise dynamic semantics of these operations, with particular reference to their actions in the presence of N inputs, may be found in Appendix A.

Needless to say, these requests can take unbounded time to return values to the "calling" program, due to their split-phase nature. The structure controller follows a simple control path to produce these results. A finite state machine implementation of this locking protocol is outlined below. The methodology below allows multiple outstanding *LOCK* and *UNLOCK* requests at any given time, queueing them for later matching requests. No busy-waiting is ever needed by the calling processor or process.

The abstract state-transitions defining these operations within the I-structure memory controller may be found in Figure 4.2 and Table 4.1. In this chart, *contents* refers to the contents of the locking cell. The right-arrow ($\rightarrow$) is used to specify the next state. The initial state is UNLOCKED.

This simple functionality could be used, for example, to implement a serializing asynchronous queue with the following code:[†]

[†]Actually, since the order of enqueuing on a lock cell is nondeterministic and not guaranteed to be FIFO (or any other ordering), the queues we are implementing are not FIFO or LIFO queues. They may be though of instead as unordered task queues.

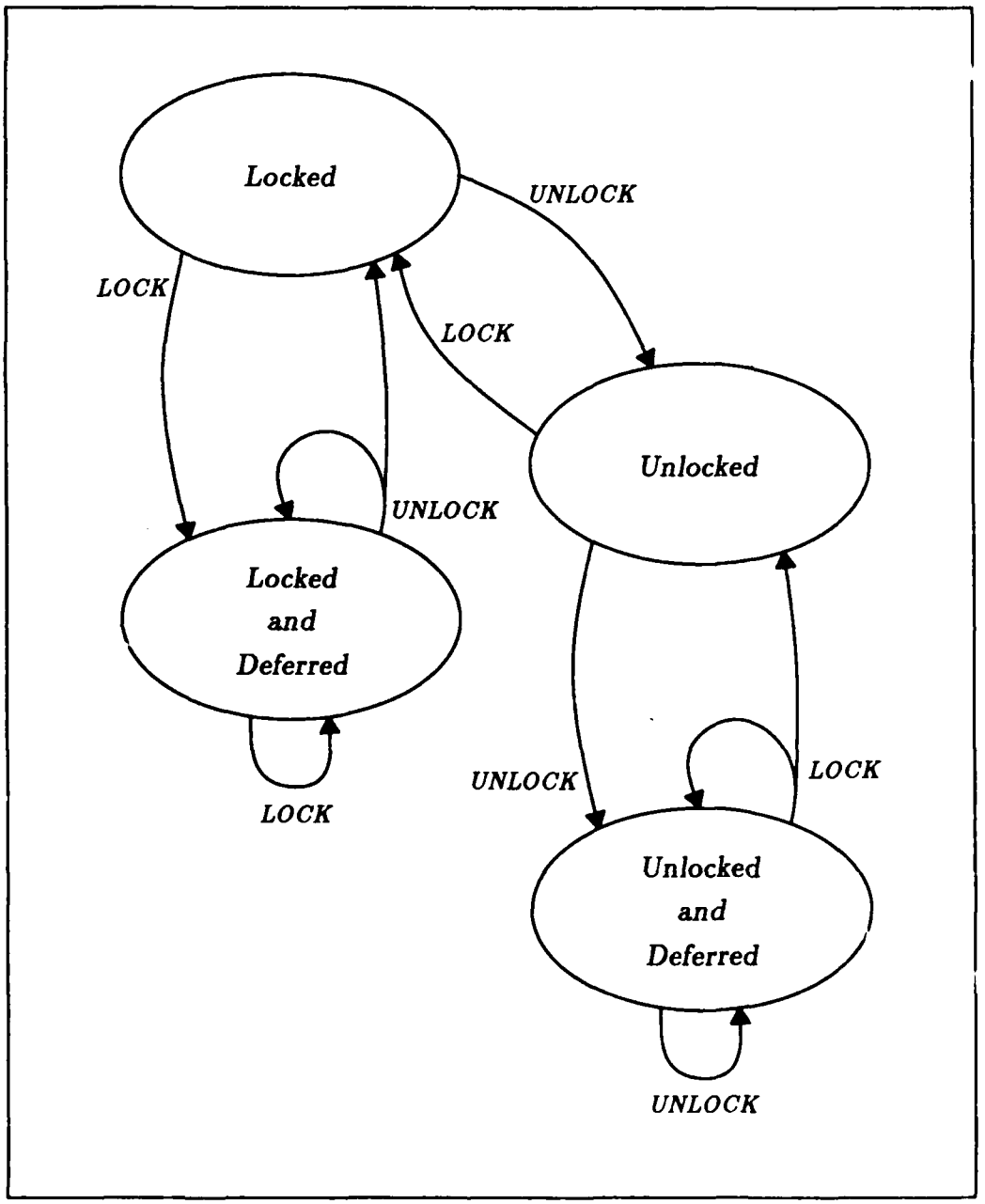


Figure 4.2: State Diagram for the Locking Protocol

STATE	LOCK (*continuation*)	UNLOCK (*value*)
LOCKED	*contents* = cons *continuation* nil → LOCKED/DEFERRED	*contents* = *value* → UNLOCKED
LOCKED/ DEFERRED	*contents* = cons *continuation contents* → LOCKED/DEFERRED	send *value* to (head *contents*) *contents* = tail *contents* if *contents* = nil then → LOCKED else → LOCKED/DEFERRED
UNLOCKED	send *contents* to *continuation* → LOCKED	*contents* = cons *value contents* → UNLOCKED/DEFERRED
UNLOCKED/ DEFERRED	send (head *contents*) to *continuation* *contents* = tail *contents* if *contents* is list then → UNLOCKED/DEFERRED else → UNLOCKED	*contents* = cons *value contents* → UNLOCKED/DEFERRED

Table 4.1: State Transitions for the Locking Protocol

```
def push element queue =
    { old_list = lock queue;
      new_list = cons element old_list
      unlock queue (gate new_list (tl new_list)) };

def pop queue =
    { old_list = lock queue;
      new_list = tl old_list;
      unlock queue new_list
      in
         hd old_list };
```

In fact, the entire lock/unlock framework outlined above can be looked upon simply as hardware support for queuing, which dataflow machines that support the I-structure model need anyway.[41, 14] With this different slant, we can see a simpler implementation of **push** and **pop**:

```
def push element queue =
    unlock queue element;

def pop queue =
    lock queue;
```

For some programs, outstanding deferred *UNLOCK* requests may not be necessary for correct operation; likewise deferred *UNLOCK* requests may be difficult to implement for some architectures.† A simplified model that correctly implements the protocol (sans outstanding *UNLOCK* requests) is presented.

The abstract state-transitions may be found in Figure 4.3 and in Table 4.2. Again in this chart, *contents* refers to the contents of the locking cell. The right-arrow (→) again is used to specify the next state. The initial state remains UNLOCKED.

STATE	LOCK (*continuation*)	UNLOCK (*value*)
LOCKED	*contents* = cons *continuation* nil → LOCKED/DEFERRED	*contents* = *value* → UNLOCKED
LOCKED/ DEFERRED	*contents* = cons *continuation contents* → LOCKED/DEFERRED	send *value* to (head *contents*) *contents* = tail *contents* if *contents* = nil then → LOCKED else → LOCKED/DEFERRED
UNLOCKED	send *contents* to *continuation*	send *error* to *continuation* → UNLOCKED

Table 4.2: State Transitions for the Simplified Locking Protocol

Some more examples will clarify the semantics of the locking operation outlined above. A function which exchanges the value in a cell with another value, using a related locking cell, might be written (quite sequentially!) as is outlined below.

†Note, however, that our implementations of **push** and **pop** given *do* require locking with deferred *UNLOCK* requests, as there is no guarantee of request ordering.

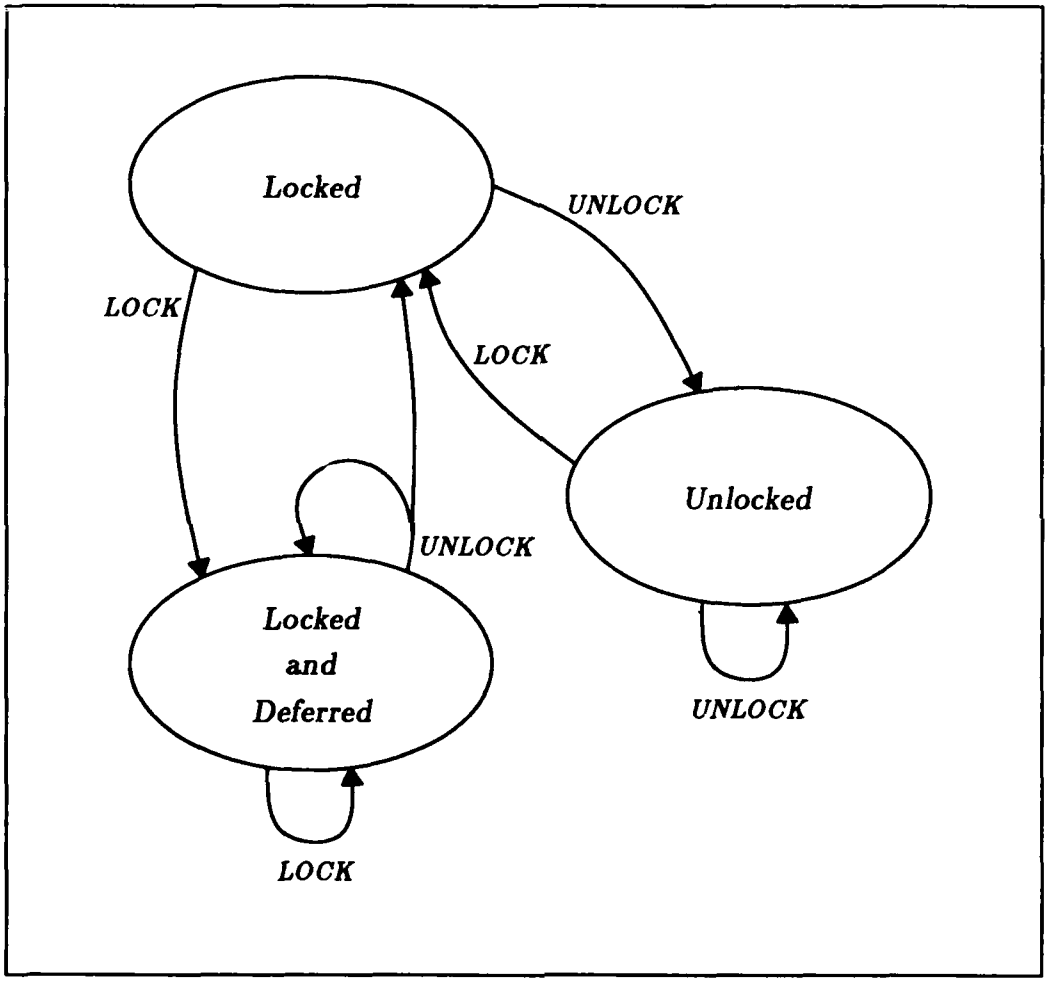


Figure 4.3: State Diagram for the Simplified Locking Protocol

```
def exchange cell new_value =
    { old_value = lock cell;
      unlock cell (gate new_value old_value)
      in
         old_value };
```

We can also implement more powerful constructs. For example, each of three processes competing to produce a best possible answer for some problem might all deposit their results in a single cell *if and only if* it represents the best result found so far. This "maximizing" function (really a sort of "compare-and-swap") could be implemented as follows:

```
def maximize cell new_value =
    { old_value = lock cell;
      unlock cell (max new_value old_value)
      in
         old_value };
```

4.4 Using New Structure Semantics in Managers

A particularly important use of locking semantics is that required to implement the manager serialization entries needed for correct implementation of resource management constructs as outlined in Section 3.3. Although approaches to this problem have been studied cursorily for other dataflow architectures,[23, 46] no solution for tagged-token dataflow systems such as Monsoon[61] have been published. Here we show an implementation using the locking constructs suggested. We depend heavily on the view of locking as a queueing mechanism, as shown above.

We deal first with the creation of managers. Given a manager creation directive

manager function

to create a new manager object and entry for a manager with no state, we need to accomplish two things:

- Create a manager serialization entry object (a *name* or *address* for that manager, in essence) to be written to by callers and read from by the manager code body.
- Wrap the appropriate nondeterministic operators around the functional code body **function** to allow it to act on incoming serialized requests.

The serialization entry is implemented as an I-structure cell used as a lock. A *manager request block* containing information about the request to the manager and space for a result is created, filled in by the requestor, and passed to the manager by entering it onto the manager's input queue (*i.e.*, by using **unlock**). A manager request block, as used in this report, is a three-cell structure (*3-tuple*), with the layout given in Figure 4.4. The requestor creates this request block, and fills in the request *type* with a symbol denoting the type of request; and the request *data*, which varies by request type. The requestor then reads from the *result* slot of the tuple. This of course relies on the implicit synchronization of I-structure memory to continue to requestor when the manager has finished processing the request.

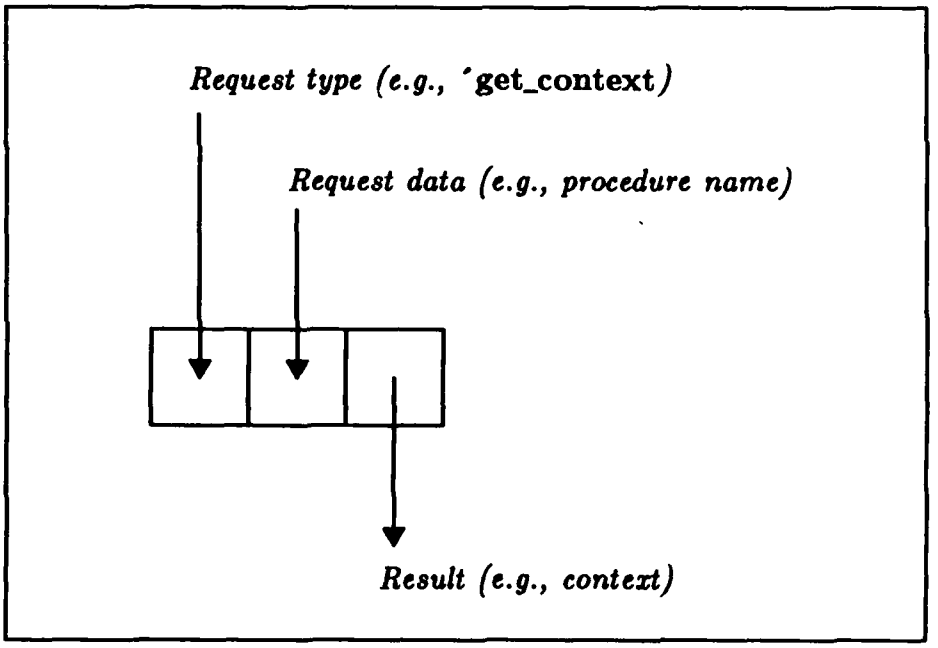


Figure 4.4: Structure of a Manager Request Block

Managers are thus created by the function

```
def manager function =
    { alive = true;
      entry = make_lock nil;
      % Loop until manager is terminated by request.
      { while alive do
          request = lock entry;
          request_type = request[0];
          data = request[1];
          % Check to see if request is for manager termination.
          next alive = request_type ≠ 'destroy_manager;
          request[2] = function request_type data }
      in
         entry };
```

which returns a serialization entry (structure cell) which can be used to pass requests to the manager code body. This structure can be seen more clearly in the abstract graph of Figure 4.5.

Managers with inherent state are created by the ID call

manager function initial_state

and can be implemented via

```
def manager function initial_state =
    { alive = true;
      state = initial_state;
      entry = make_lock nil;
      % Loop until manager is terminated by request.
      { while alive do
          request = lock entry;
          request_type = request[0];
          data = request[1];
          % Check to see if request is for manager termination.
          next alive = request_type ≠ 'destroy_manager;
          result, next state = function state request_type data
          request[2] = result }
      in
         entry };
```

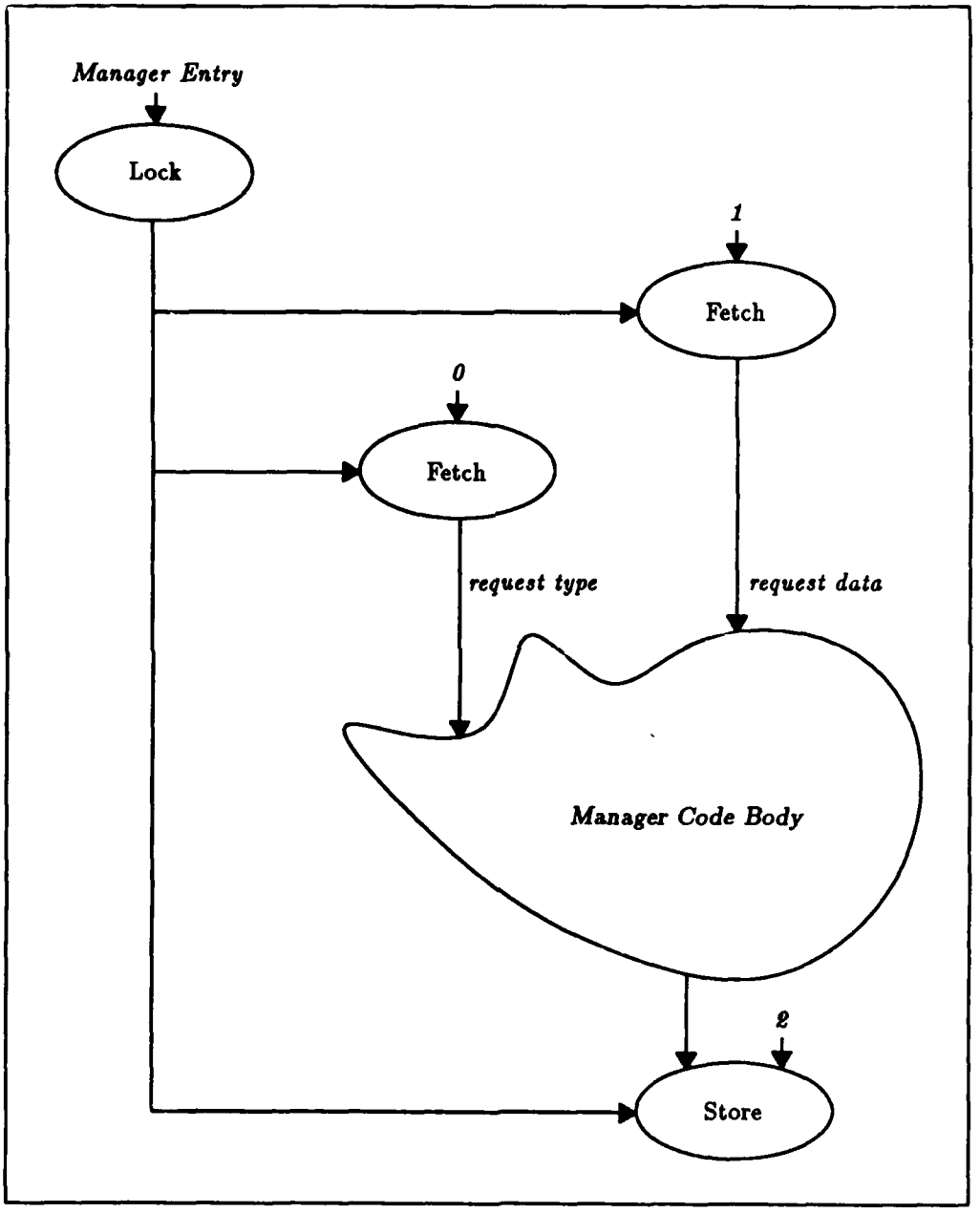


Figure 4.5: Manager Abstract Graph with No State: Loop Body

This function is identical to the previous implementation of **manager**, except that it circulates manager state information as returned by the manager code body. The abstract graph for this approach can be seen in Figure 4.6.

The sole remaining abstract graph to present is, of course, the structure for calling managers (*i.e.*, the **use** primitive). This algorithm, as outlined at the beginning of this section, can be coded in ID in the quite straightforward manner below; the resultant graph is outlined in Figure 4.7.

```
def use manager request_type request_data =
    { request = request_type, request_data, ?;
      _, _, result = request;
      unlock manager request
      in
         result };
```

4.5 Fine Points of Manager Implementation

These simple ID programs and their related graphs gloss over several important fine points of manager use and construction, however. In particular, the codes given above might entice us to construct our application manager in the following manner:

```
def application_manager_code_body request_type data =
    if request_type == 'get_context
      then Allocate context object large enough for function data
      else Deallocate context object data;

system_application_manager = manager application_manager_code_body;
```

A second thought about the structure of managers, however, makes it clear that this will not work, as the application manager code body itself will need to perform a function application to call **application_manager_code_body**. Clearly, application managers (and probably allocation managers) are primitive in some special way. Either they must be provided by a lower level of the operating system, as in the dataflow virtual memory technology of the author and Steele,[69] or a primitive function application style must be used. The latter

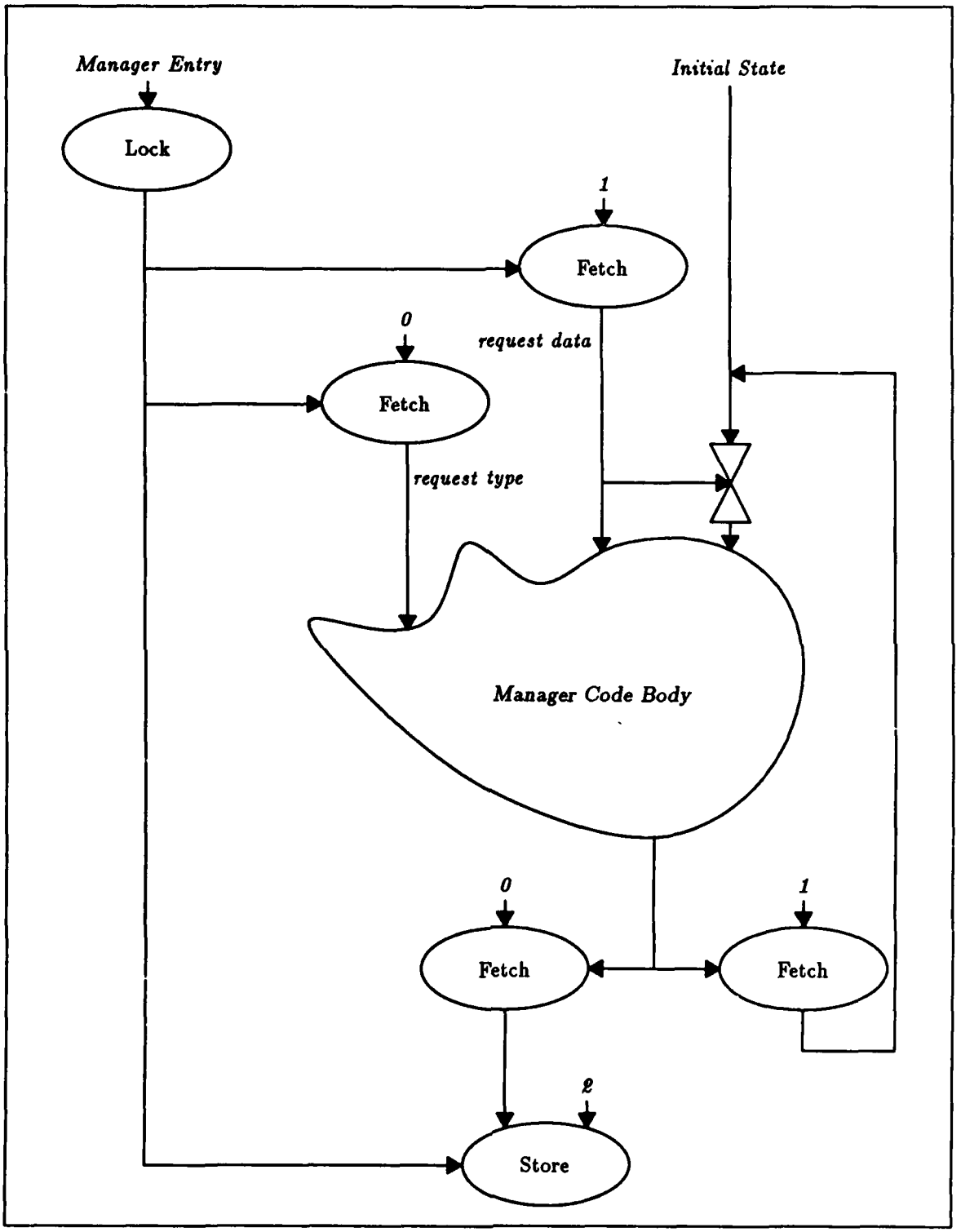


Figure 4.6: Manager Abstract Graph with State: Loop Body

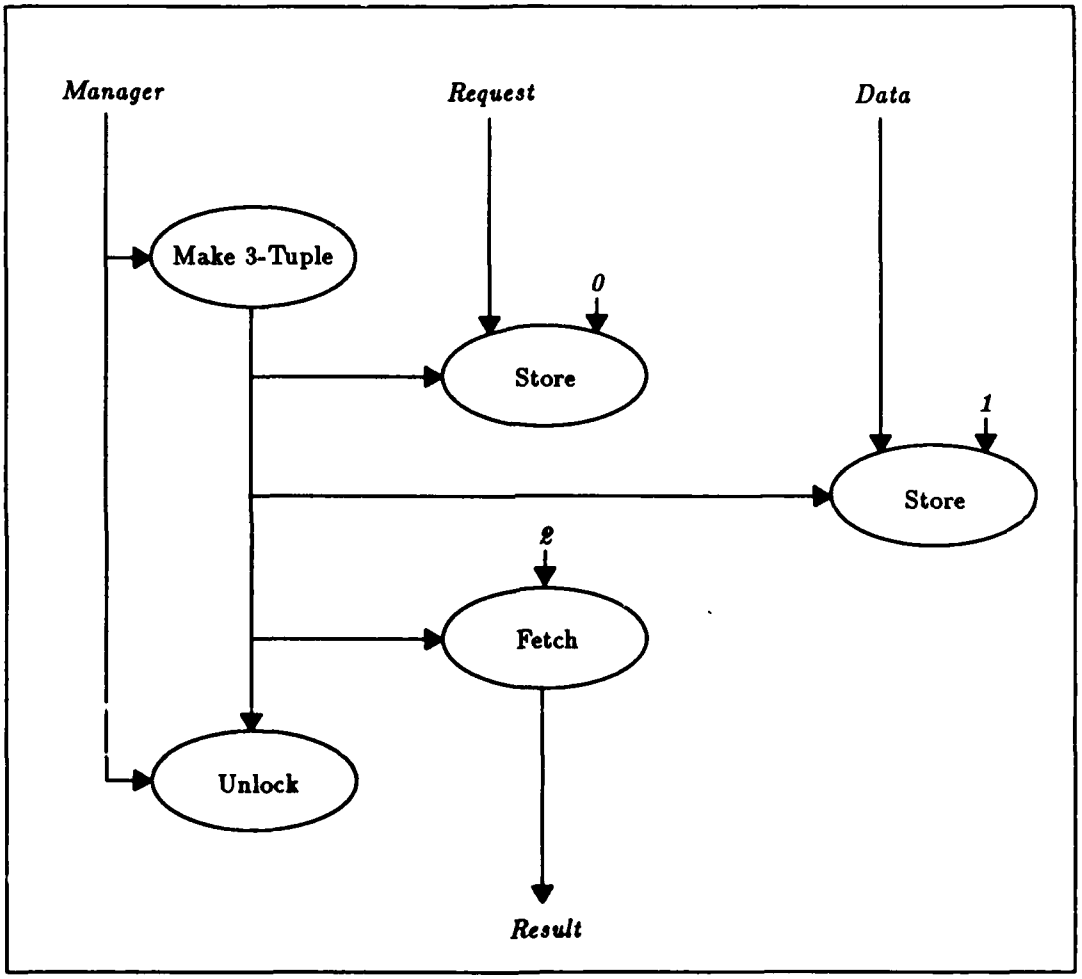


Figure 4.7: Manager Use Abstract Graph

structure was chosen for use in this report, with special managers written for function application and allocation. Managers to filter requests, however, were implemented exactly as proposed above.

There are, however, other possible problems with the approach to manager implementation given, revolving primarily around issues of fairness and deadlock avoidance. In particular, since the locking primitives presented give no guarantee of queuing order, the resulting managers do not necessarily arbitrate "fairly" between requests. However, order of request arrival is not necessarily at all related to the fair order of request handling in a dataflow system, even if the requests were issued by the same task or procedure. This is due to the fact that instruction scheduling in a dataflow machine is determined solely by data dependence issues, with no other ordering control in general.[†] Fairness (and possibly resultant minimized critical execution paths) is a difficult issue in dataflow architectures, and is discussed in the work of Steele[68] and Barth[17, 16, 18] in the context of managers and locking primitives.

Deadlock avoidance is an important aspect of any operating system support, and is particularly difficult in general managers given ID non-strict structure semantics. For example, consider the following situation:

- A manager's loop body (as outlined above) is evaluated sequentially (*i.e.*, one-bounded loops as defined by Culler[28] are in place).
- The first manager request block retrieved from the manager's input queue has a request type of '**get_context** (*i.e.*, to request allocation of a function application context), but the request data cell of the block has not yet been filled in.

Given this simple situation, no further manager requests will be handled by this manager until the request data (*e.g.*, function to call) has been determined and passed to the manager. At best this will tend to lengthen the critical execution path of the application; at worst a deadlock will occur (for instance, if a function call is necessary to determine the request data!).

Unfortunately, simply unrolling the manager code body loop is not a complete answer. Although it lessens the possibility of such a deadlock, it does not decrease that probability to zero. The solution used in this report was to avoid queuing requests until the request type and data had been determined and stored (*i.e.*, by making request queuing *strict*).

[†]Exceptions include the **gate** operator discussed in Section 3.2, input/output serialization discussed elsewhere,[66] and the locking primitives outlined above.

This works for our purposes (application and allocation) because we can clearly state our strictness needs; all request types and data are non-compound (non-aggregate) data. If, however, in general managers, request data is composed of I-structure references, strictness becomes more complex to insure. Barth explores this problem further in his work.[18]

The last problem of real manager implementation is the central structure of managers given. Not only will this approach have a tendency to cause network hotspots in a real multiprocessor machine, but a sequentialization of function applications can occur if the critical path of the function application manager itself is long compared to the critical path of the average procedure used. Solutions to this problem generally involve multiple managers for a given operating system function, and are beyond the scope of this thesis. It is expected that Barth will touch on this problem as well in his doctoral work.[18]

4.6 Dynamic Binding

The sole remaining architectural feature we need in order to realize the language structures of Chapter 3 is dynamic binding support. There are two standard approaches to realizing dynamic free variable binding within languages. The first, and simplest, is called *deep binding*.[71] In a deep-binding scheme, a *binding stack* (or *list*) is constructed and used to map free variable names to values. This table is then passed dynamically from procedure to procedure (*e.g.*, as a function argument). Associative lookup is then used to find the dynamic value of a free variable. Thus dynamic variable binding schemes can be implemented within lexically scoped languages. For example, in ID, a source-to-source method could be used to translate the following:

```
% Declare the variable x to be dynamically scoped.
@dynamic x;

% Apply f to a with x bound to value.
def withx value f a =
    { x = value
      in
        f a };

% Increment input value y by current value of x.
def inc y =
    x + y;
```

to the equivalent code for a lexically-scoped environment

```
def withx value f a bindings0 =
    { bindings1 = cons ('x, value) bindings0
      in
        f a bindings1 };

def inc y bindings0 =
    (lookup_dynamic_value 'x bindings0) + y;

def lookup_dynamic_value variable nil = nil
|   lookup_dynamic_value variable (symbol, value) : tail =
    if variable == symbol
      then value
      else lookup_dynamic_value variable tail;
```

In other words, we add an implicit argument to every piece of user code to pass along the current dynamic scope information. Dynamic bindings are achieved by adding to the front of the binding stack; lookups are linear matching searches through that stack.

Other schemes for accomplishing dynamic scoping architectures have been considered and used in non-dataflow contexts. The most popular is probably the *shallow-binding* approach of the Lisp Machine world.[67] This hybrid scheme for implementing dynamic binding within a naturally lexically- and globally-scoped environment stores current dynamic variable values in variables of global scope, and then uses a binding stack (as outlined above) to remember *old* dynamic values. This permits fast dynamic binding of variables, *and* fast lookup of

current dynamic variable values, *but it depends on sequential code execution.* Since global values are accessible to all code executing, at any given time this scheme supports only a single binding for a dynamically-bound variable. Therefore it is unacceptable for a parallel execution environment such as a dataflow execution paradigm.[†]

Our proposal for dynamic binding under dataflow paradigms is deep binding, but with a twist. In a normal LISP deep-binding scheme, the binding list is built with one binding element per bound variable.[‡] Then a binding list might appear as in Figure 4.8.

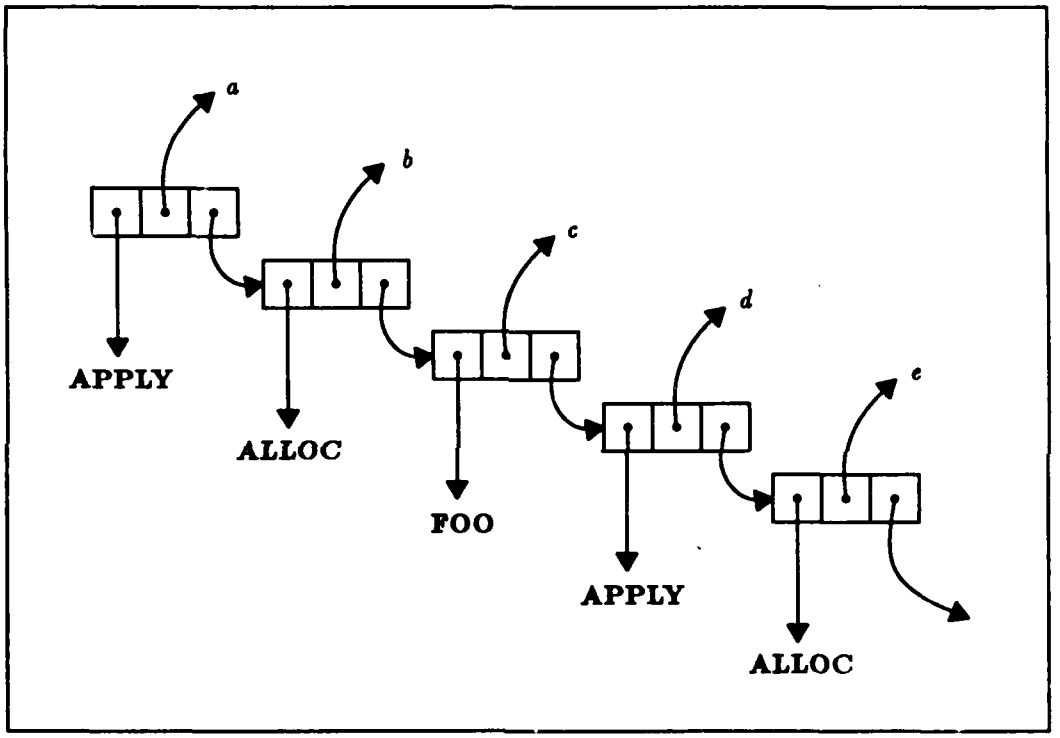


Figure 4.8: Simple Deep Binding List

This structure is quite simple to create, although it can be expensive to search (*e.g.*, for variable binding lookup). Worse, it fails to take advantage of the fact that in our use of dynamic binding (at least), we will *always* be binding the application and allocation manager symbols (here represented as **APPLY** and **ALLOC**) at the same time. Knowing this, we might represent binding lists in a manner which made these variable names *implicit* and assumed, as in Figure 4.9.

[†]Actually, the Lisp Machine environment is parallel in that it supports multiprogramming. The shallow-binding scheme used in Lisp Machine environments forces expensive task switches to correctly switch shallow binding stacks.

[‡]Where in LISP a "binding element" might be two *conses*, a 3-tuple serves well in ID.

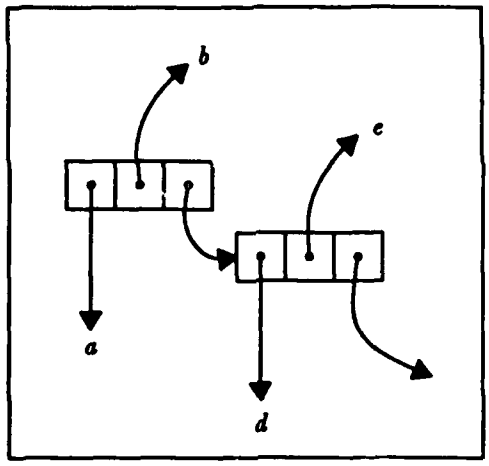


Figure 4.9: Deep Binding List with Implicit Variable Names

In this model, each time we open a new binding level we assume that we are binding both **APPLY** and **ALLOC**, the former in the first cell of the binding block and the latter in the second cell, with the third cell reserved to link to previous binding levels. This makes lookup quite fast, as it is done simply by a tuple fetch on a lexical variable.

Unfortunately, this does not fully solve the problem. The binding of **FOO** to c in Figure 4.8 was not preserved in Figure 4.9. Assuming that **FOO** is a variable that we wish to dynamically bind only sporadically, requiring another slot in the implicit variable name scheme has unacceptable overhead. The approach we use instead is a hybrid of the two schemes, which allows implicit variable names for commonly- (and jointly-) bound variables, with explicit names for other variables. Figure 4.10 shows how we would structure the bindings of Figure 4.8 under this hybrid scheme.

Regardless of binding list implementation, all of these methods use lexical scoping to implement dynamic scoping through the use of binding lists. It is interesting to note the parallels between this use of added lexical arguments and the use mentioned by the author in his work on implicit I/O ordering, a related problem.[66] There might, in fact, be some unification of the two systems.

4.7 Summary

In this chapter we have provided abstract architectural modifications to the general tagged-token dataflow design to allow managers as defined in Chapter 3, to be used in operating

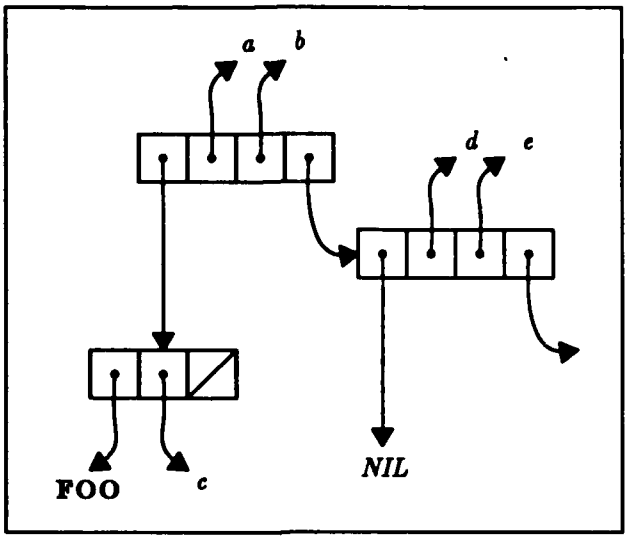


Figure 4.10: Hybrid Deep Binding List Structure

system modes (*e.g.*, function application and memory allocation) as well as in user codes requring support of state-based computation. In addition, we have given extensions to dynamic dataflow designs to support the **blackboard** style of nondeterministic computation as well as the dynamic variable binding mechanism we use to define tasks. In the following chapter we will put this work to use.

Une maison est une machine à habiter.
— LE CORBUSIER, *Vers une architecture*

Chapter 5

Experiments and Results

We will journey in this chapter step-by-step through an example program, and see if our new approach to speculation adds the expressive power we sought with the efficiency we need. Measurements of efficacy based on parallelism profiles, total instructions executed and critical execution path length are used.

5.1 An Example

In this section we add to the repetoire of examples started in Chapter 2 the Eight-Puzzle game search described by Nilsson.[59] We choose to use the Eight-Puzzle game because Nilsson displays so many different search strategies for the game, with clear discussions of the repercussions of various approaches.

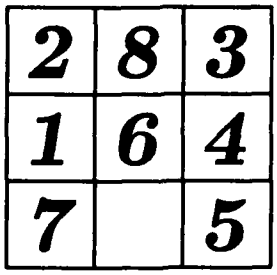


Figure 5.1: The Eight-Puzzle: A Starting Position

The Eight-Puzzle game is a familiar hand-held "brain teaser" in which eight numbered square tiles (*e.g.*, numbered 1 to 8) are arranged in some fashion on a three-by-three square grid, as in Figure 5.1. One square is empty; thus a neighboring tile can be slid into that

slot. The object of the game is to arrange the numbered tiles on the board such that the central slot is empty, while the numbered tiles are arranged in order around the center, as in Figure 5.2. Though the solution for the game is not obvious, the rules of play (and therefore approach to programmatic solution) are straightforward.

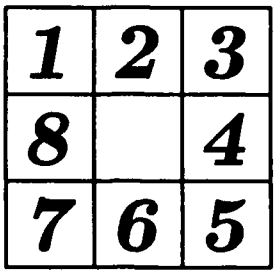


Figure 5.2: Solution Position for the Eight-Puzzle

In our solution, we will eschew the view of moving numbered tiles into the blank space, but rather view the problem in the light of swapping the *blank* "tile" with a neighboring *numbered* tile. Thus we can avoid computing move legality (*i.e.*, avoid checking to see if we're moving a tile onto another tile). From this point of view, there are a limited number of moves possible from each position based on the current location of the blank space. For each position $i \in [0,8]$ we can compute a number encoding each of the possible moves (up, right, down and left) as a bit in a four-bit number (*i.e.*, *up* = 1, *right* = 2, *down* = 4 and *left* = 8). Figure 5.3 graphically displays the possible moves for each puzzle cell; the number in the upper-left corner is the cell number, while the lower-right number encodes the possible moves for that cell.

We will represent a position in the Eight-Puzzle game as a 9-element zero-based vector, with offset i corresponding to cell i of the game, and with 0 marking the blank cell. Thus the ID vector

2	8	3	1	6	4	7	0	5

encodes the position shown in Figure 5.1. The following ID code implements the puzzle solver:

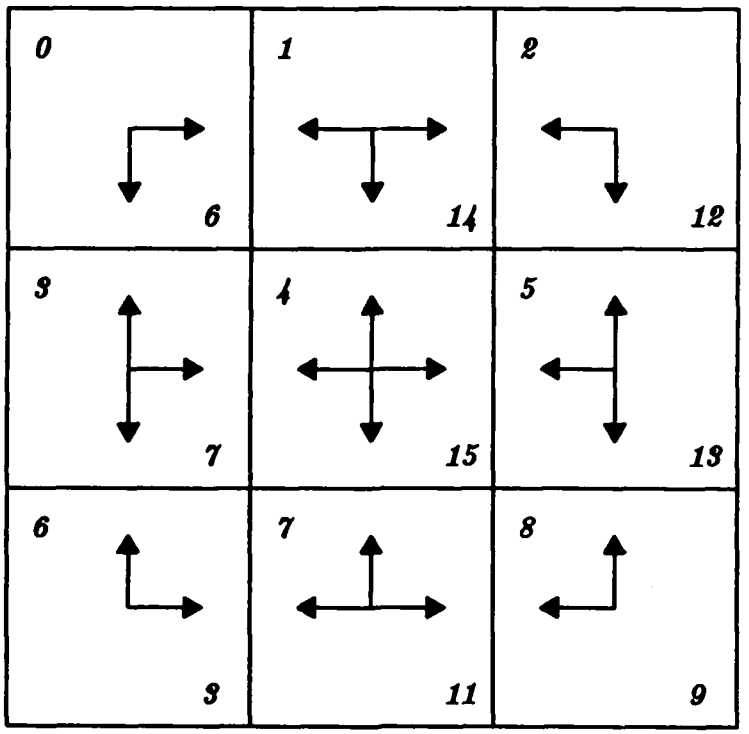


Figure 5.3: Possible Move Map for the Eight-Puzzle

```
% Encode the possible move map of Figure 5.3.
def move_map =
    {vector (0, 8)
     | [0] = 6 | [1] = 14 | [2] = 12
     | [3] = 7 | [4] = 15 | [5] = 13
     | [6] = 3 | [7] = 11 | [8] =  9 };

% Compute the possible blank "tile" moves from a current
% position of the blank tile.
def compute_possible_moves blank =
    { def possible_moves direction offsets =
        if direction == 0
          then nil
          else { rest = (possible_moves (direction ≫ 1) (tl offsets))
                 in
                   if (move_map[blank] ∧ direction) ≠ 0
                     then (cons (hd offsets) rest)
                     else rest };
      in
        possible_moves 8 (-1 : 3 : 1 : -3 : nil) };
```

%%% This code is continued on page 114.

%%% *This code is continued from page 113.*

```
% Compute the new board situation from the current board
% situation and blank tile location and a possible blank tile move.
def compute_new_board board blank move =
    {vector (0, 8)
     |      [i] = board[i + move] when i == blank   % Swap into blank tile.
     |      [i] = 0 when i == blank + move          % New blank tile.
     ..     [i] = board[i] };

% Find the blank tile in a board position.
def find_blank_tile board =
    { def find board i =
          if board[i] == 0
            then i
            else find board (i + 1);
      in
        find board 0 };

% Compare two board layouts for equality.
def compare_boards board1 board2 =
    { same? = true
      in
        { for index ← 0 to 8 do
              next same? = same? and (board1[index] == board2[index])
            finally same? } };

% Solve the puzzle, starting from a particular position.
% Do not search more than search_depth for solution.
def solve_the_puzzle starting_board search_depth =
    solve_puzzle nil search_depth (board, (find_blank_tile starting_board));

% The solution board layout.
def solution =
    {vector (0, 8)
     | [0] = 1 | [1] = 2 | [2] = 3
     | [3] = 8 | [4] = 0 | [5] = 4
     | [6] = 7 | [7] = 6 | [8] = 5 };
```

%%% *This code is continued on page 115.*

```
%%% This code is continued from page 114.

% Find all successor positions of the current board position.
def successors board blank =
    { def successor move =
          compute_new_board board blank move, blank + move;
      in
         map_list successor (compute_possible_moves blank) };

% The heart of the solver: Take a board and blank position, list
% of positions seen so far, and search depth count. Stop if
% we've seen this position, or it's the solution, or we've
% reached the maximum search depth. Else recursively search.
def solve_puzzle boards_seen depth (board, blank) =
    if compare_boards board solution
      then boards_seen % Return series of boards searched.
      else if depth < 0 or member? compare_boards board boards_seen
            then terminate nil
            else speculate
                  (solve_puzzle (cons board boards_seen) (depth - 1))
                  (successors board blank);
```

The heart of the program is the code block **solve_puzzle**, which recursively speculates on possible next board positions (*e.g.*, movements of tiles into the blank tile position). At each node, it checks to see if the game has been successfully solved; if so, the program has finished. If, on the other hand, the program has generated a position already searched, *or* if the maximum search depth has been reached, it terminates the search branch. Otherwise the program continues searching recursively. A typical depth three search tree that might be computed by the program above in finding the solution, from Nilsson,[59] is in Figure 5.4. The reader may wish to try re-coding **solve_puzzle** (and its helping functions!) using only the simple **blackboard** construct of Chapter 2; we really have gained some expressive power through the use of **speculate**!

It is also important to note that we have picked a relatively small search problem for this first example. The branching factor for the Eight-Puzzle search tree averages 1.75, creating a tree growth which may be found in Figure 5.5. We will also examine statistics for longer searches.

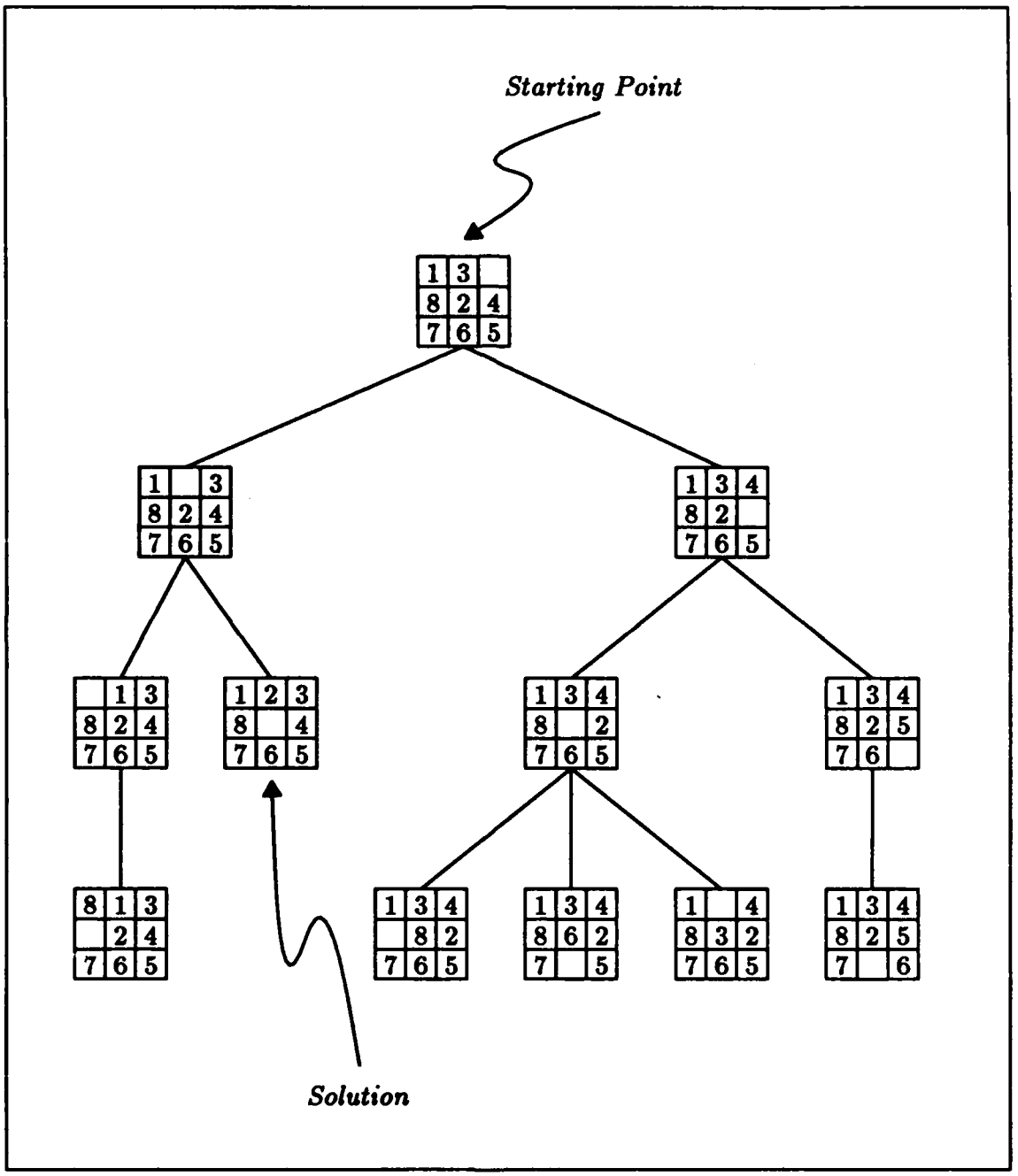


Figure 5.4: Typical Depth Three Search Tree for Solve_Puzzle

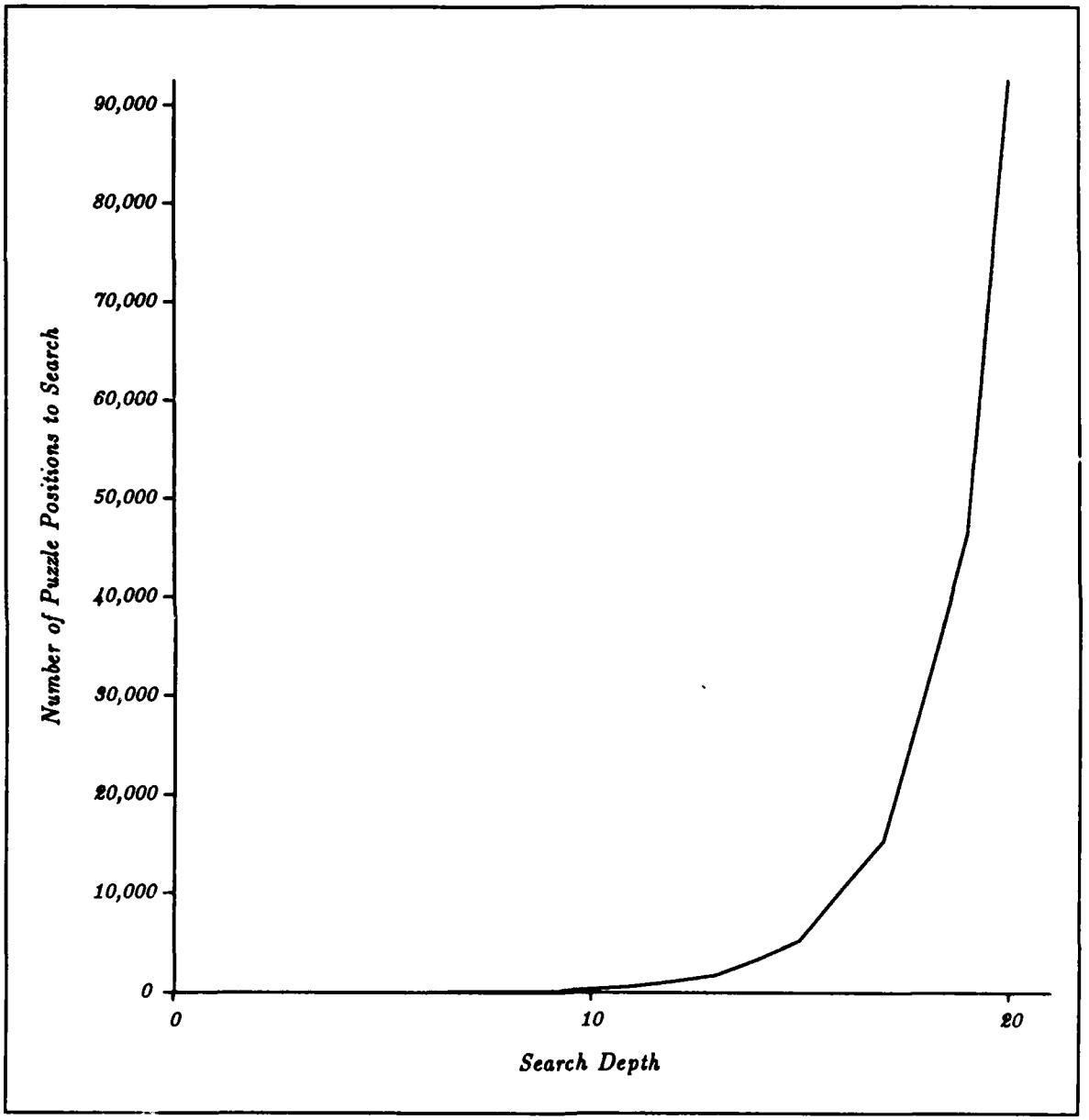


Figure 5.5: Growth of Eight-Puzzle Search Tree by Search Depth

5.2 Speculation Features

In order to gauge the efficiencies of the approach to speculation taken by this report, the Eight-Puzzle code described above was used. A particular problem (that of Figure 5.4), with a known solution composed of two moves, was given to the **solve_puzzle** procedure. The code for the puzzle solver was used exactly as presented above, but with five different styles of speculation. Each study was instantiated simply by varying the definition of **speculate**. This section reports the results of these runs.

The first approach to speculation to be presented is the simple, maximally parallel EXPLOSIVE SEARCH. The ID code to implement this style of search is quite straightforward:

```
def speculate function list =
    { result = 1d_i_array (0, 0);
      { for elt ← list do
            answer = function elt;
            if novalue? answer
              then {}
              else { result[0] := answer }}
      in
         result[0] };
```

This implementation of **speculate** simply calls the input function on every element of the input list, returning the result which is written into the one-element vector **result** first. This code ignores questions of error-handling and propagation raised and answered in the context of the proper solution for the implementation of **speculate**, in Chapter 4. Nevertheless, it provides a concrete baseline with which to compare other approaches to solving speculative parallelism problems.

Executed for a search depth of three, **solve_puzzle** used with this implementation of **speculate** examines every possible subtree at each choice point in the search, despite the fact that the solution is found at a search depth of two. This results in the ALU parallelism profile in Figure 5.6, consuming 55,183 instruction cycles with a critical execution path of 5,516.

A BACKTRACKING SEARCH approach to implementing **speculate** linearizes the solution search path, much as in logic languages such as PROLOG. The implementation used for

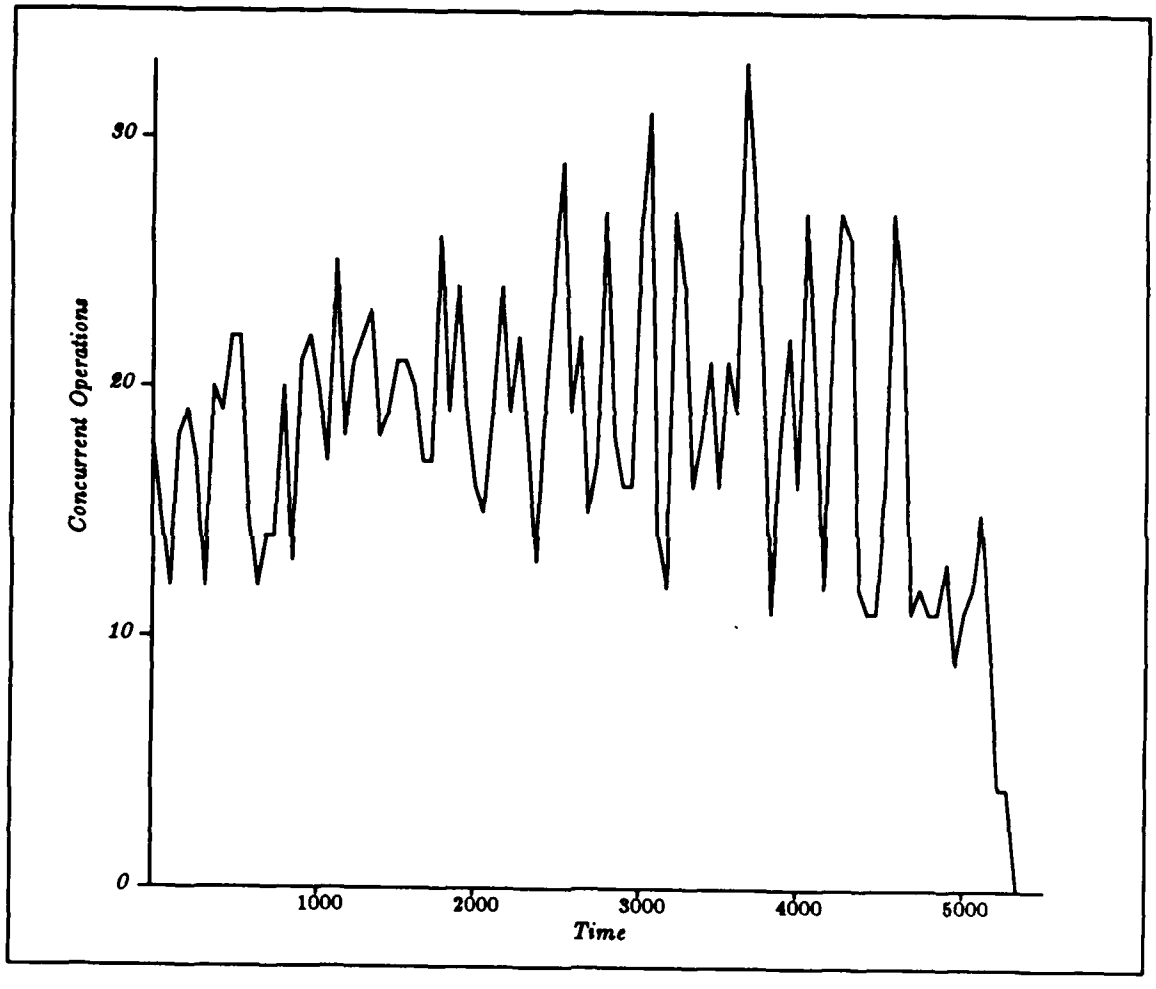


Figure 5.6: ALU Parallelism for Explosive Puzzle Search

this study directly realizes depth-first backtracking, by checking solutions one at a time and sequentializing future search paths based on completion of the last. The implementation in ID is again straightforward:

```
def speculate function list =
    if list == nil
      then terminate nil
      else { answer = function (hd list)
             in
                if novalue? answer
                  then speculate function (tl list)
                  else answer };
```

As expected, for a search depth of three, we see a smaller number of instructions to find a solution: 21,565. Suprisingly, we also see a shorter critical path, of 3,478. This is accounted for by the higher average search depth of the explosive approach, combined with the idiosynchratic way that results are collected in that implementation. The resulting ALU parallelism for BACKTRACKING SEARCH may be found in Figure 5.7.

At last we come to a real form of speculation, wherein the machinery of speculation control (*e.g.*, dynamic managers filtering application and allocation requests) are in place. However, the results for EXPLOSIVE SPECULATION are in some sense a measure of the pure *overhead* of speculation control, since in this measure we do *not* do any task termination. In other words, this implementation executes all possibilities, collecting the first non-failing result, but *not* terminating any running tasks. The results, displayed in Figure 5.8, are terrible as one might expect. The critical path was 55,017, with 685,433 instructions executed to search three levels deep in the search space.

But these results are to be expected, since EXPLOSIVE SPECULATION is simply EXPLOSIVE SEARCH with speculation control overhead. For this reason, we will not explore EXPLOSIVE SPECULATION more deeply.[†]

Once we return termination to the picture, better results are found. In particular, SPECULATION WITH TERMINATION is the implementation of speculation as presented in Chapter 3,

[†]Besides, the simulation system (Gita) which was used fails after simulating more than several seconds of a tagged-token machine!

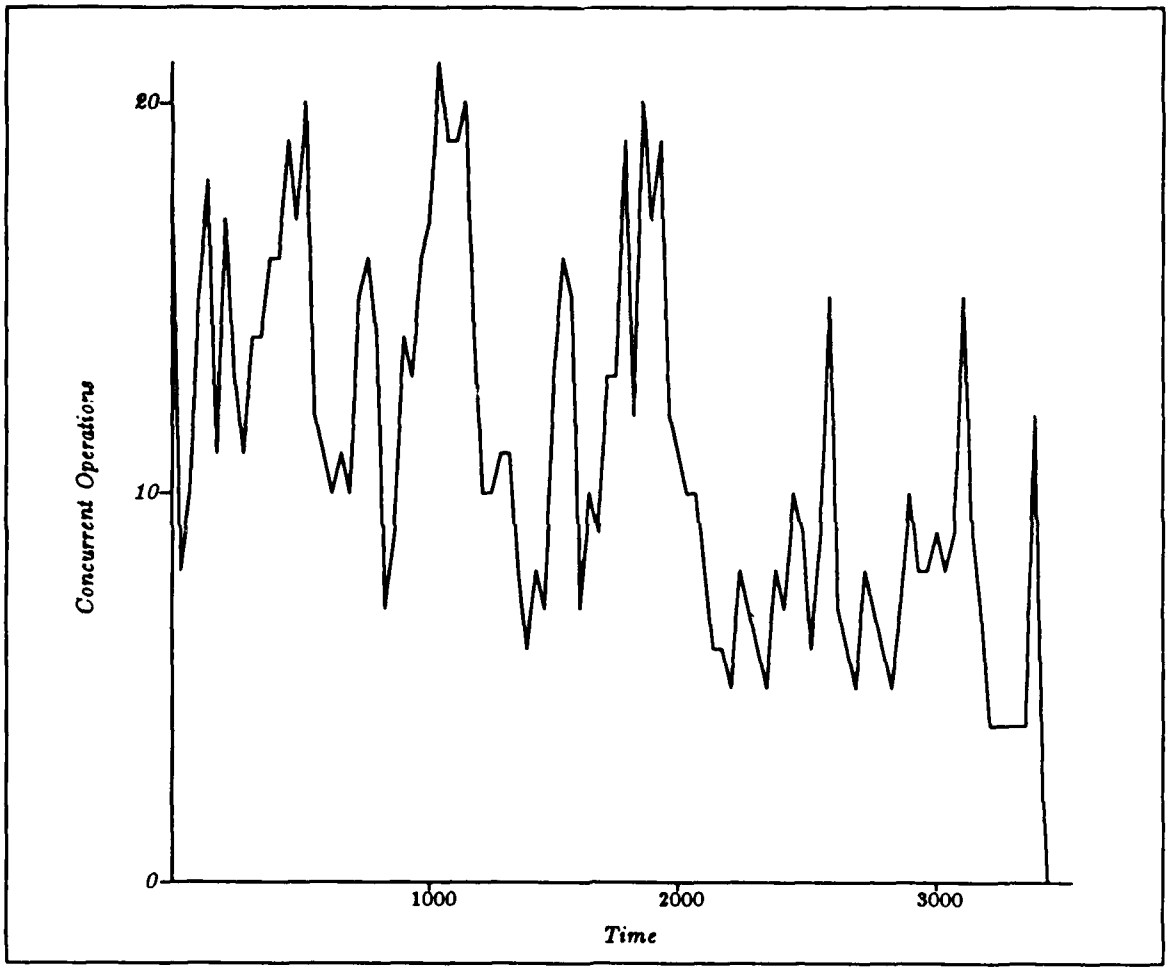


Figure 5.7: ALU Parallelism for Backtracking Puzzle Search

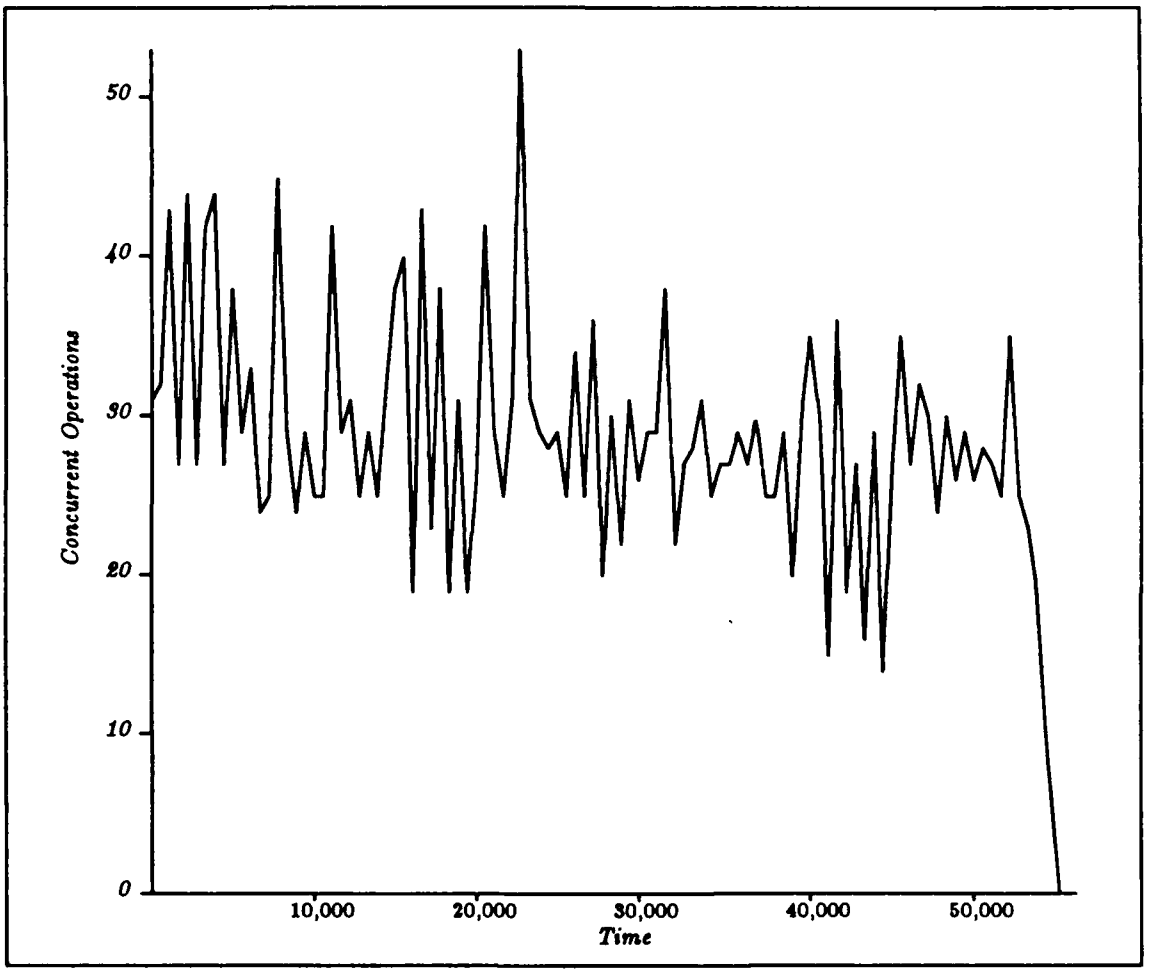


Figure 5.8: ALU Parallelism for Speculative Puzzle Search

but with a special implementation of **terminate** that simply returns ℵ (but does not actually request termination of the current task). After the first valued response is returned, termination of all other interspeculative tasks is requested. For this approach, with a depth-tree search, we find ALU parallelism as in Figure 5.9, with 123,614 instructions executed during a critical path of 9,909 instructions.

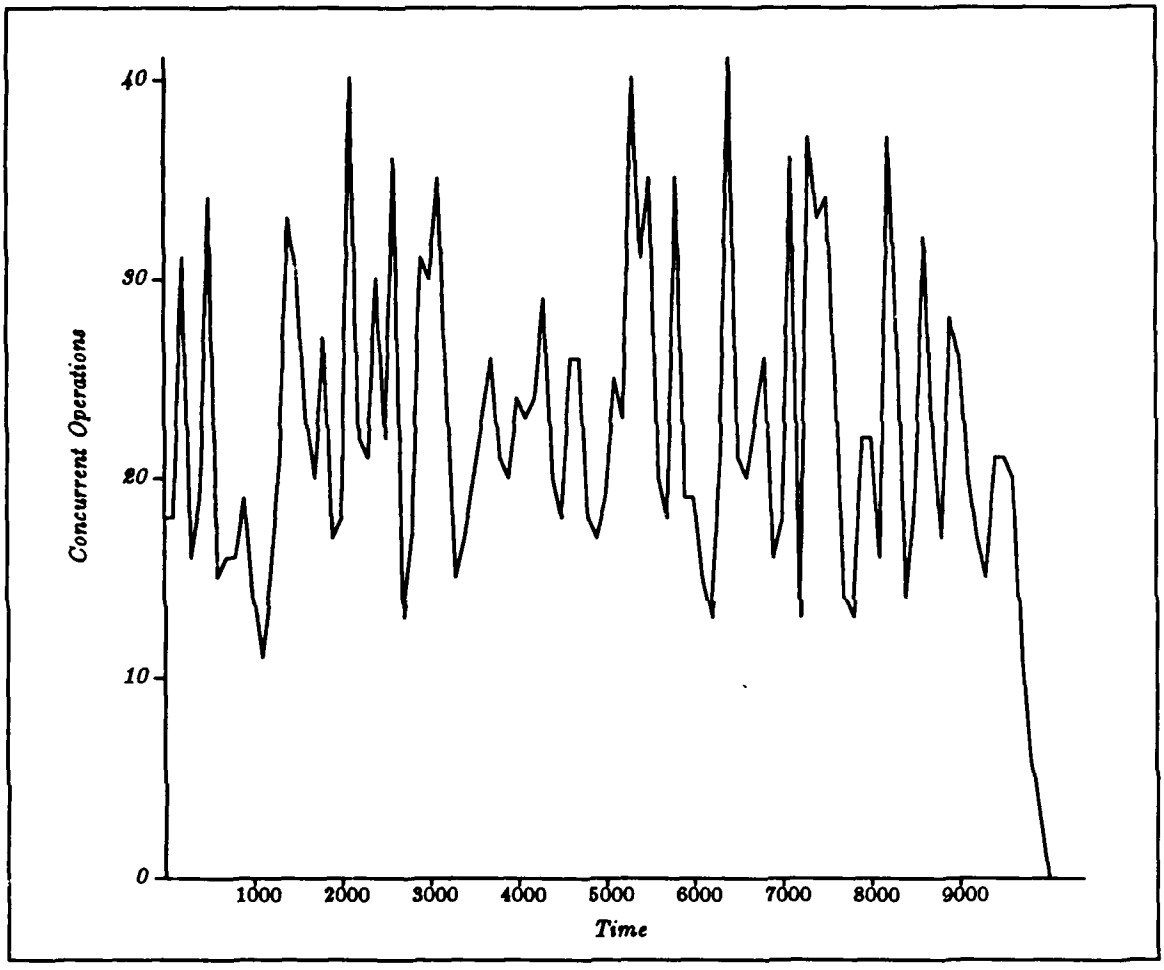


Figure 5.9: ALU Parallelism for Speculative Puzzle Search with Termination

The last step, SPECULATION WITH SELF-TERMINATION, implements the full non-prioritized **speculate** and **terminate** as implemented in Chapter 3. Here **terminate** is still defined to return ℵ, but additionally has the side effect of requested termination of the current task. Now we see parallelism as displayed in Figure 5.10, with only 97,120 instructions in a critical path of 7,944.

Table 5.1 summarizes these results.

Surprisingly, Table 5.1 appears to show that speculation is a losing proposition; simple

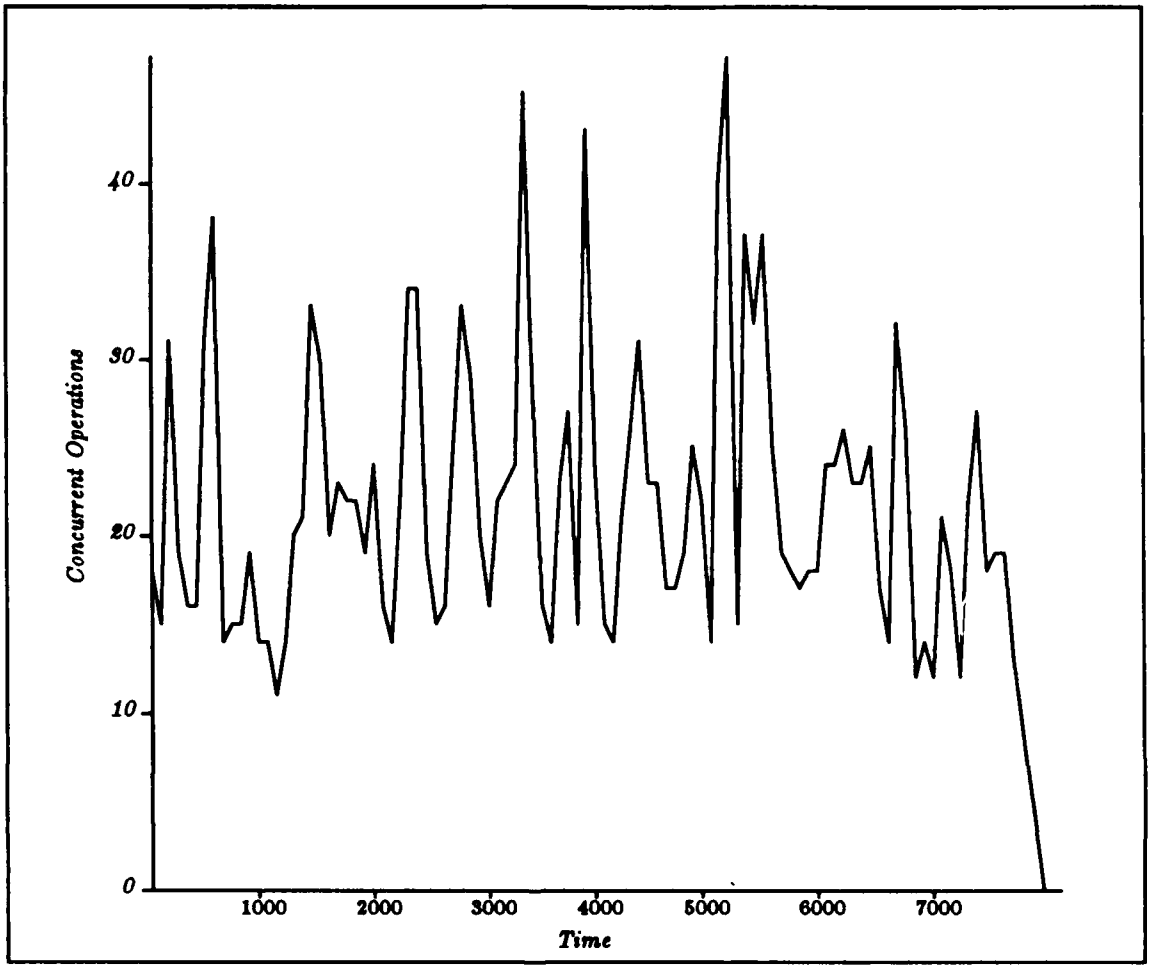


Figure 5.10: ALU Parallelism for Speculative Puzzle Search with Self-Termination

STRATEGY	INSTRUCTIONS	CRITICAL PATH
EXPLOSIVE SEARCH	55,183	5,516
BACKTRACKING SEARCH	21,565	3,478
EXPLOSIVE SPECULATION	685,433	55,017
SPECULATION WITH TERMINATION	123,614	9,909
SPECULATION WITH SELF-TERMINATION	97,120	7,944

Table 5.1: Puzzle Solution Performance with Search Depth = 3

BACKTRACKING SEARCH wins handily even over the "complete" solution, SPECULATION WITH SELF-TERMINATION. This is due to the inherent overhead of the speculative approach. In deeper searches, we find much better arguments for speculation with control. Table 5.2, for example, presents simulation results for the four interesting speculation approaches, with the same problem and a search depth of five.

STRATEGY	INSTRUCTIONS	CRITICAL PATH
EXPLOSIVE SEARCH	218,714	20,310
BACKTRACKING SEARCH	62,218	11,435
SPECULATION WITH TERMINATION	123,614	9,909
SPECULATION WITH SELF-TERMINATION	97,120	7,944

Table 5.2: Puzzle Solution Performance with Search Depth = 5

Already speculation wins over other approaches on a critical-path basis, although the total work is higher in SPECULATION WITH SELF-TERMINATION than in BACKTRACKING SEARCH. However, at more reasonable search depths (for the given problem), speculation wins hands down. Table 5.3 presents results for the same problem and inputs, but with a search depth of eight.

STRATEGY	INSTRUCTIONS	CRITICAL PATH
EXPLOSIVE SEARCH	1,493,342	128,883
BACKTRACKING SEARCH	388,220	78,166
SPECULATION WITH TERMINATION	123,614	9,909
SPECULATION WITH SELF-TERMINATION	97,120	7,944

Table 5.3: Puzzle Solution Performance with Search Depth = 8

Needless to say (and as expected), the overhead of speculation is non-trivial, and must be amortized over the execution of a search. If there is only shallow search, or the amount of work to be computed at each node is quite small, the overhead of speculation will overpower the actual useful work. However, at deeper search depths or with more complex computations, speculation control overhead provides the encapsulated support we want with efficient solutions. Figure 5.11 and Figure 5.12 show this result by displaying the total number of instructions executed and critical execution path, respectively, of the Eight-Puzzle solution code run by various speculation methods, and over the three search depths.

We must note that the parallelism of the graphs and counts presented here does vary somewhat from what has been published in previous accounts of research in dynamic dataflow execution paradigms. This is in general due to our accounting for resource management overhead, and to the particular implementation of our resource management. As implemented for this study, no resource manager can have a critical path less than 22 instructions; calling a manager, going through the manager code body loop, and receiving a result incurs a cost of at least 200 instructions. Clearly these numbers can be improved upon; prototypical primitive managers for the Monsoon machine[61] do better than this by a wide margin.

However, a more important problem is brought forward by the overall shapes of the graphs presented here. They have a tendency toward longer critical paths not seen in other work on the tagged-token architecture. This is caused by the serial handling of application manager requests; for small procedures this approach tends to serialize the program unnecessarily. Solutions to this problem exist, however; *proactive* managers (which push as much of the work ahead of requests as possible) and *multiple* managers for a given resource are considered promising. These problems, and those discussed in Section 4.5, are currently being investigated by Steele[68] and Barth.[17, 18]

5.3 Prioritized Speculation

We turn now to prioritization of interspeculative tasks. At first blush it might seem that the version of the puzzle solver presented above implements an equal-priority search (*i.e.*, **priority** 1 in the syntax given in Chapter 3) of the descendants of a choice point. In fact, this is not the case; each interspeculative task uses as many system resources as it needs without regard to the usage of its "brother" interspeculative tasks.

Instead, we may want to control interspeculative tasks among each other, balancing the usage of each against the usage of the others. If we use the version of **speculate** with priorities given in Chapter 3, and do not specify priorities (*i.e.*, just use the puzzle solving code as specified above) we get a search with the priorities of all interspeculative tasks equal; thus no task's requests for system resources (*e.g.*, I-structure memory and function application) are allowed to heavily outweigh any other task's usage. This then implements a *mostly best-first* search, since subtrees of each search point will share the balanced priority

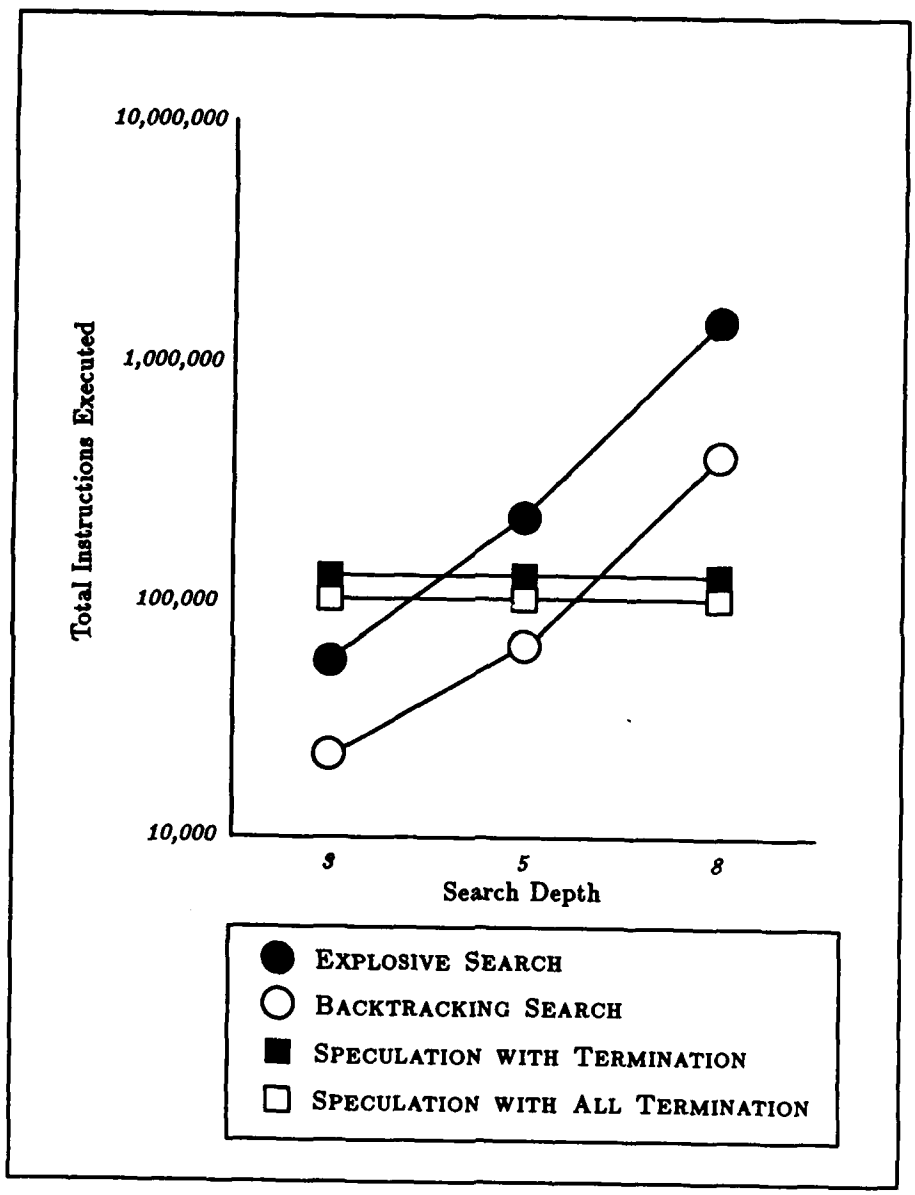


Figure 5.11: Puzzle Solution Performance: Total Instructions by Depth

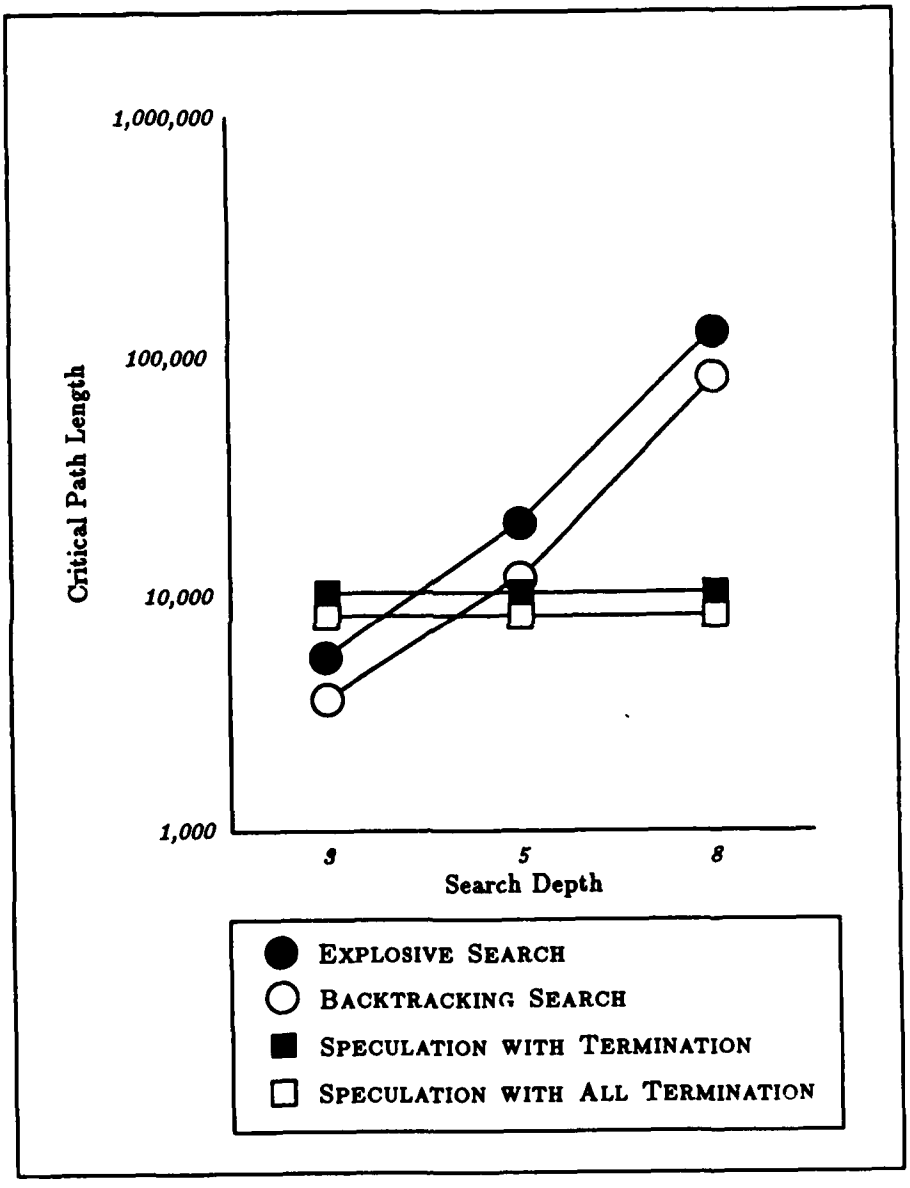


Figure 5.12: Puzzle Solution Performance: Critical Path by Depth

of their parents.

If instead we chose to rewrite **solve_the_puzzle** and **solve_puzzle** as in the following code:

```
% New puzzle solver with mostly-depth-first search.
def solve_the_puzzle starting_board search_depth =
    solve_puzzle nil search_depth
                 ((starting_board, (find_blank_tile starting_board)), 1);

% Solve_puzzle rewritten to set task priority.
def solve_puzzle boards_seen depth ((board, blank), prio) =
    { priority prio
      in
        if compare_boards board solution
          then boards_seen
          else if (depth < 0) or (member? compare_boards board boards_seen)
                 then terminate nil
                 else { new = successors board blank;
                        priorities = compute_priorities new
                        in
                          speculate (solve_puzzle (cons board boards_seen)
                                                  (depth - 1))
                                    zip2 successors priorities }};

% Compute a descending list of priorities for left-to-right mostly-depth-first.
% For example, (compute_priorities ≪list of length 3≫) → (4 2 1).
def compute_priorities list =
    { l = length list;
      n = 1;
      result = nil
      in
        { for i ← 1 to l do
            next result = n : result;
            next n = 2 * n
          finally result }};
```

we would in effect have a *mostly depth-first* search, in which subnodes of a search point to the "left" (examined first) are allowed more system resources than other interspeculative tasks. Similarly, we could implement a *best-first search* using a variant of Nilsson's suggested heuristic[59]

$$f(n) = d(n) + W(n)$$

for determining the relative "goodness" of a particular point in the search tree (where f computes the relative merit of a particular subnode n, $d(n)$ is the depth of the subnode n in the search tree, and $W(n)$ is the number of tiles "out of place" in reference to the correct placement of tiles in the solution). That may be coded in ID as:

```
% New puzzle solver with mostly-best-first search.
def solve_the_puzzle starting_board search_depth =
    solve_puzzle nil search_depth
                 ((starting_board, (find_blank_tile starting_board)), 1);

def solve_puzzle boards_seen depth ((board, blank), prio) =
    { priority (goodness board depth)
      in
        if compare_boards board solution
          then boards_seen
          else if (depth < 0) or (member? compare_boards board boards_seen)
                 then terminate nil
                 else speculate (solve_puzzle (cons board boards_seen) (depth - 1))
                                (successors board blank) };

% Compute a priority for each task based on a heuristic measure.
% Returns higher priorities for positions more likely to be in the solution path.
def goodness board depth =
    { total = depth
      in
        { for i ← 0 to 8 do
            next total = if board[i] == solution[i]
                         then total + 1
                         else total
          finally total }};
```

As expected for our three-deep search (with a solution found at depth two) these codes show different behaviors. The mostly best-first approach wins, followed closely by the most depth-first (due to the fact that the ordering of the search happened to match the ordering of the solution), followed by the breadth-first search.

Unfortunately, all three solutions had instruction counts and critical paths at least an order of magnitude higher than the non-prioritized versions of Section 5.2. In those codes, the extra overhead of speculation control was minimal, with a few instructions to check for termination interceding before resource usage was granted. In our implementation of

prioritization, multiple queues of "ready to run" application and allocation requests, scanned many times for balancing purposes, increased the overhead of the approach over the usable limit (indeed, over the amount of "real" computation to solve the problem).

5.4 Summary

In this chapter we saw that our new speculative constructs extend the expressive power of ID in a useful fashion for writing searching types of programs. Although not entirely successful from the standpoint of system efficiency, we saw how dynamically defined tasks can be controlled on a prioritized basis as well. In the following chapter, we will look at other kinds of programs in ID which might benefit from a speculative approach. We will finish with an exploration of possible future directions of the work: support of alternative approaches to speculation, and general uses of the task control constructs we have built.

An ensampull yn doyng ys more commendabull
þen ys techyng oþer prechyng.
— JOHN MIRK, *The Festyuall*

Chapter 6

Conclusions and Future Directions

Chapters 1 and 2 outlined a problem in the solution of typical Artificial Intelligence codes; although taking advantage of the speculative parallelism of these programs seems to be the natural course of mapping searching programs to parallel hardware, the dataflow paradigm (and others!) require some control over the resultant explosive tree of execution. Prioritization and termination (particularly the latter) are often the basic levels of control needed by these codes. Chapter 2 presented a simple-minded methodology for such control by extending the ID language using a straightforward nondeterministic construct (the **blackboard**). We saw that this construct, while theoretically quite powerful, was expensive to use in both execution and programming. A better approach to speculation, using the construct **speculate**, was suggested to encapsulate the nondeterministic behaviors in a simple-to-understand structure.

Chapters 3 and 4 proceeded to give implementation details for **speculate**. While uncovering the details of resource management for the ID language under dataflow execution, we saw that some support for dynamic scoping of names was desirable, particularly for precisely defining the idea of a "task" so that task termination could be implemented. Propagation of $\aleph$ to support the collapsing of tasks undergoing termination was explored to ensure that important dataflow invariants could continue to be supported.

Finally, in Chapter 5, we saw a finished application using **speculate** and related new ID language constructs, along with some measurements of the expense of using those structures. We saw that the **speculate** and **terminate** forms successfully encapsulate the nondeterministic structures needed to control explosive execution tree growth, while preserving the

dataflow execution invariants for correct termination of dynamic execution trees. Nevertheless, we have not imposed on the programmer a need to explicitly specify algorithm parallelism, or directly manipulate task structures.

In this chapter we will expand on the usefulness of the basic structures supporting speculation in order to assist other speculative programming styles.

6.1 Speculative Convergence

We saw in Chapter 5 that there are critical path elongating problems with our approach to prioritized speculation. Other kinds of non-prioritized speculation control also show promise, however. A common type of loop expression in numeric code implements *convergence.* For example, the fragment

```
{ while error > tolerance do
   ...
   next error = ...
   ...}
```

is often used to express the looping computation which continues until **error** is less than or equal to **tolerance**. While this correctly implements the intent of computing until some convergence is reached, under the dataflow execution paradigm it causes a single bottleneck disallowing parallel calculation of the loop body. This is because we do not know whether an execution of the loop body may be begun until the previous value of **error** has been computed by the previous run of the loop body. This common occurrence causes low-parallelism points in otherwise high-parallelism codes, for example the Simple code.[28] Figure 6.1 displays the ALU parallelism of the Simple code over time; although the maximum parallelism of the code is over 6,000 instructions, and even the average parallelism is greater than 500, there are several times during execution where the parallelism bottoms out (becomes one).

The usual solution for this problem in extant ID codes is to replace the convergence-test of a loop with some set number of iterations, for example

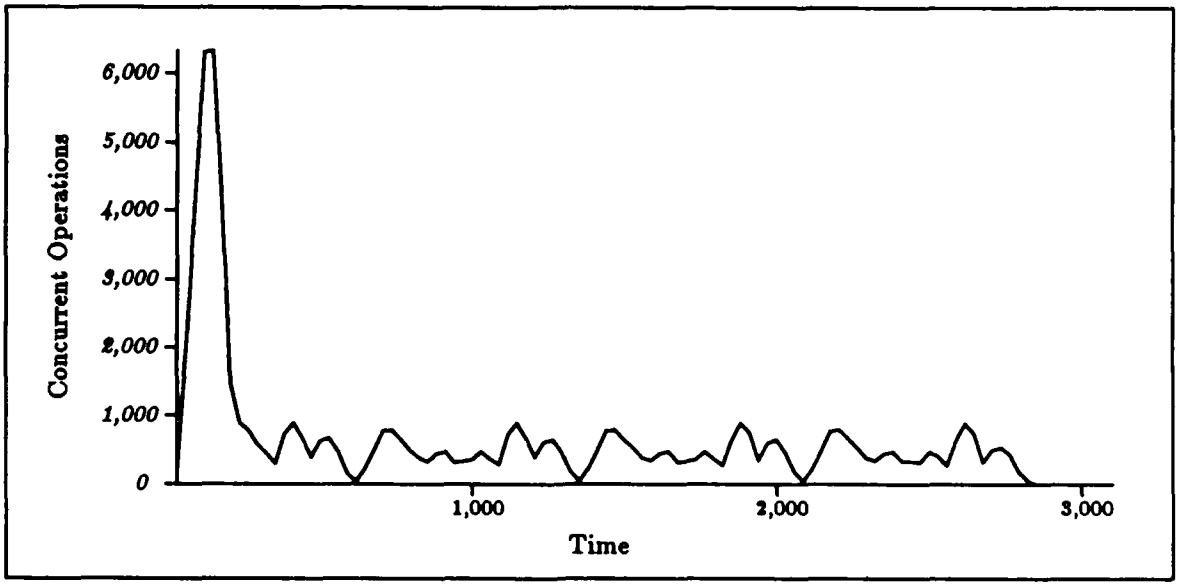


Figure 6.1: ALU Parallelism Over Time for the Simple Code

```
{ for iteration_number ← 1 to 5 do
    ...
    next error = ...
    ...}
```

For some codes with weak inter-loop-body dependencies, this style of programming relaxes the strict sequentialization of loop instantiations and allows parallel execution of the loop body for each iteration. However, there are two problems with this solution:

- The code no longer declares the intent of the programmer. Besides confusing the point of the loop, the number of iterations may not be appropriate for the calculation of the loop series to convergence.
- Worse, the number of iterations may *overshoot* the number necessary for convergence within the tolerance the programmer wishes to express. Therefore, unnecessary (and perhaps expensive) calculations may be performed, increasing both total instruction count and critical path length.

Speculation allows a middle ground. What we would really like to specify for series calculations with few dependencies between loop bodies is the tolerance computation for the loop control (as in the first example), but with some allowance for execution of "future" instantiations of the loop body. In addition, we could use the control features developed in this report to stop execution of these "future" threads when the convergence condition is

finally met.

We can code this in a style analogous to that of **speculate** in the following way:

```
% Speculation-style series convergence.
def speculative_converge function initial_state tolerance speculation =
    { result = i_array (0, 0);
      state = initial_state; count = 1;
      termination = make_lock false;
      still_looping = i_array (0, 0);
      still_looping[0] = true;
      tasks = i_array (1, maximum_iterations);
      { for i ← 1 to ntasks do tasks[i] = true };

      def task state i =
          { old_apply = application_mgr;
            old_alloc = allocation_mgr;
            in
              { application_mgr = manager speculation_mgr old_apply tasks i;
                allocation_mgr = manager speculation_mgr old_alloc tasks i;
                error, new_state = function state;
                call terminate_manager application_mgr answer;
                call terminate_manager allocation_mgr answer;
                old_lock_value = lock (gate termination answer);
                result_determined? = not (novalue? answer);
                in
                  if result_determined? and (old_lock_value == false)
                    then { if error < tolerance then result[0] = answer;
                           call remove_other_tasks tasks i ntasks
                           call unlock termination true
                           still_looping[0] := false;
                           in
                             new_state }
                    else { unlock termination old_lock_value
                           in
                             state }}};

      { while still_looping[0] bound speculation do
              next state = task state count;
              next count = count + 1 }
      in
        result[0] };
```

This code works in exactly the same way as our earliest working **speculate** without pri-

oritization nor task self-termination; we control tasks via the allocation and application managers. The function **speculative_converge** expects as arguments a function (which takes some internal state and returns a new error and state), an initial state, a minimum error tolerance, and the amount of speculation to allow. This speculation amount is enforced simply by setting the loop bound of the loop to that amount, which restricts the unrolling of the loop (maximum co-executing loop bodies) to the control amount.

6.2 Future Directions

We have shown how parallel tasks to implement parallel speculative search can be implemented in a consistent manner within the ID language and for tagged-token dataflow machines. Specification of parallelism is implicit in our approach, as are the details of task control and termination. The implementation is brought to the ID language level through three simple constructs: **speculate**, **terminate** and **priority**, only the first of which is actually necessary for a complete solution.

Since the specification and control of speculation is entirely within the language, we could imagine having other versions of **speculate** with different semantics. We have seen in this chapter that other semantics may be applied to the **speculate** form to achieve differing results. Highly explosive or depth first implementations of **speculate** were displayed in Section 5.2. We could also imagine implementing a simple speculation semantics *à la* PROLOG:

```
def speculate function list =
    { alive = true;
      element : tail = list;
      { while alive and tail ≠ nil do
            next element : next tail = tail;
            terminated = i_array (0, 0);
            terminated[0] := false;
            control_mgr = manager set_termination terminated;
            answer = function element;
            alive = terminated[gate 0 answer]
        finally if alive
                    then answer, tail
                    else 'fail };

def terminate x =
    use control_mgr 0 0;
```

This implementation of **speculate** returns the first non-failing (in the PROLOG sense) result of applying **function** to elements of **list** left-to-right; it also returns a tail of the list so that further backtracking search can be done by re-calling **speculate**. Further research into useful definitions of **speculate** and related functions (including the failed experiment in prioritization) would greatly extend this work.

It would also be useful to look for other "choke points" on execution graph growth (*i.e.*, other than function application and memory allocation). For example, it might be useful to compile all (or certain) loops with schemas that check for termination in the enclosing tree of tasks, and stop looping if termination is found.

In addition, a different syntactic structure for speculation might be helpful. Separating the *generation* and *execution* of speculative threads would make certain problems easier to state. The problems we have looked at in this report assumed that the branching factor at each decision point is static during speculation itself; in fact, there are problems for which we would like to add more interspeculative tasks during the process of speculation. A syntactic structure more like CLU's *iterators*[53] might be a good solution.

Finally, this report has developed a definition of *tasks* (dynamically enclosed regions of the execution graph) that makes intuitive sense, despite the parallel nature of computation even within a task. Moreover, the approach to termination we have taken leaves the concept well-

defined, rather than the careless descriptions of termination found in most operating system manuals.[†] We could use these better-defined tasks to solve other speculative problems (for example, in which more than one, or all, solutions of a speculative search are required) or non-speculative problems requiring task support (such as operating system task control).

The future ain't what it used to be.
But then, it never was.
— LEE HAYS & THE WEAVERS

[†]See, for example, the description of the `kill` command or function under the Unix operating system.[27]

Appendix A

Dataflow Operator Strictness in ℵ

In this appendix we develop precisely what we mean by strictness in ℵ, in terms of the exact behavior of dataflow operators given the presence and intended usage of ℵ, as well as in compiler behavior for generating compound, self-cleaning program graph schemata.

The precise dynamic semantics of tagged-token dataflow actors have been expounded on in various sources[9, 11, 61, 13]; we choose a variant of Traub's syntax[73] for its clarity in expressing exceptional cases. In this expression, a *token* comes in several types for transmitting messages between the instructions of a dynamic dataflow graph. Five token types are found in the model:

- $\ll$DATA$, c, i, q, p, v \gg$ denotes a general data token targeted as input to an ALU (general processor) instruction. The dynamic *context*, or instantiation, of a graph addressed by this type of token is specified by the pair (c, i) for *context number* and *iteration*. The particular *instruction* (address, for example) is referred to by q, while p specifies the *port* (or operand/argument number) of the incoming datum. v carries the actual data to be sent to the instruction.

- $\ll$FETCH$, A, c, i, q \gg$ denotes a request to fetch the contents of a synchronized memory cell (I-Structure). This request to fetch the contents of the cell named A (*e.g.*, at address A) is handled by a *structure control* processor rather than a general instruction processor.[†] The result of the fetch is to be sent to the instructions specified by the

[†]The general instruction machinery may or may not be exactly the same as the structure controller machinery. Both models have been assumed in the literature.

destination list of instruction q (*i.e.*, q_{dest}), in the dynamic context (c, i) as noted previously.

- $\ll$STORE$, A, c, i, q, v \gg$ denotes a request to store the value v into the I-Structure cell named A. A *signal* signifying completion of the action is then requested to be forwarded to instruction q in context (c, i).

- $\ll$LOCK$, A, c, i, q \gg$ carries a structure memory *LOCK* request for reading and locking the memory cell named (*e.g.*, at address) A. After successfully locking the cell, the value read is then returned to the instructions specified by instruction q (*i.e.*, to q_{dest}) tagged for context (c, i).

- $\ll$UNLOCK$, A, c, i, q, v \gg$ denotes a structure memory controller request to *UNLOCK* the memory cell named A. Upon successful unlock, the value v is written into the cell for the next *LOCK* request to read. Upon completion, a request acknowledgement is forwarded to the destinations of instruction q tagged for context (c, i).

Instructions, as addressed by the q portion of tokens, are defined to specify (in general) an *operation* (q_{op}), an optional *constant* (q_{const}) and a list of *destinations* (q_{dest}). After executing the operation q_{op} on the values specified by the input tokens (and optionally the constant q_{const}), the processor generates output tokens directed to instructions specified by q_{dest}. Thus each element of the list q_{dest} specifies a (q', p') pair (destination instruction, destination port). We now have the basic language with which to specify the dynamic semantics of our library of dataflow actors, particularly with reference to $\aleph$ inputs.

By far the majority of operators, such as arithmetic instructions, fall into the broad category of binary arithmetic operators, as typified by the addition instruction +. The semantics of + are simple:

Table A.1: Behavior of Operation +

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v_1 \gg$	**if** $v_1 = \aleph \vee v_2 = \aleph$
$\ll$DATA$, c, i, q, 2, v_2 \gg$	**then** $\forall (q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', \aleph \gg$
	else $\forall (q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', v_1 + v_2 + q_{const} \gg$

Here we see that strictness in $\aleph$ is directly required by the semantics; a $\aleph$ input directly causes a $\aleph$ output. Even simpler are unary operators, which are trivially strict in $\aleph$. A good example of a unary operator is the `Identity` instruction, which simply replicates its input v to its destinations:

Table A.2: Behavior of Operation `Identity`

Input:	**Output:**
$\ll \text{DATA}, c, i, q, 1, v \gg$	$\forall (q', p') \in q_{dest} \Rightarrow \ll \text{DATA}, c, i, q', p', v \gg$

Most all tagged-token instructions fall into the broad category of unary or binary arithmetic-like operators, and are defined to be strict in $\aleph$. More interesting instructions, with necessarily more complex semantics, fall into two general categories:

- *I-Structure* operations, or dynamic program arcs, need special treatment; and
- *Encapsulation* structures, including conditionals, function calls and loops must be carefully examined.

We begin with scrutiny of I-Structure operations. The basic I-Structure operation cycle is:

1. A general purpose processor issues an `I-Structure Fetch` instruction, causing a "Fetch" request to be sent to a structure memory controller.
2. A structure memory controller either satisfies the request (if the data has been written), or defers the request until the data is available.
3. A general purpose processor issues an `I-Structure Store` instruction, causing a "Store" request to be sent to a structure memory controller.
4. Any deferred structure fetch requests are "undeferred" (*i.e.*, outstanding fetches on the same address) are satisfied.

This basic sychronization mechanism provides a methodology for creating dynamic arcs in program graphs. Therefore, we need to propagate $\aleph$ values across this arcs correctly.

The `I-Structure Fetch` instruction is basically unchanged, except for strictness in its structure (S) and offset (o) arguments:

Table A.3: Behavior of Operation I-Structure Fetch	
Input:	**Output:**
$\ll$DATA, $c, i, q, 1, S\gg$	**if** $S = \aleph \vee o = \aleph$
$\ll$DATA, $c, i, q, 2, o\gg$	**then** $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', \aleph\gg$
	else $\Rightarrow \ll$FETCH, $S_{base} + o + q_{const}, c, i, q\gg$

If in fact a request is sent to a memory control subsystem, then that system will respond (upon availability of data). We *explicitly* require our memory control units to be able to store $\aleph$; therefore we can simply send back whatever value is stored in the memory:

Table A.4: Behavior of Operation Structure Controller Fetch	
Input:	**Output:**
$\ll$FETCH, $A, c, i, q\gg$	$\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p',$ contents$(A)\gg$

The **I-Structure Store** instruction is complicated in two ways. First, since we have set a limit of two input tokens (*i.e.*, two input ports) to any given operation, we need to compile a store request into two stages. First, the structure base address of structure S is added to an offset o by a **Form Address** instruction to form a new structure address; then the result of this instruction is used by the actual **I-Structure Store** instruction to send a value store request.

The **Form Address** actor is quite simple; it is very similar, in fact, to the + instruction in action and strictness in $\aleph$:

Table A.5: Behavior of Operation Form Address	
Input:	**Output:**
$\ll$DATA, $c, i, q, 1, S\gg$	**if** $S = \aleph \vee o = \aleph$
$\ll$DATA, $c, i, q, 2, o\gg$	**then** $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', \aleph\gg$
	else $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', S_{base} + o\gg$

However, **I-Structure Store** instructions *must not* be strict in the value to be stored (v), since future fetches on the same location must fetch a $\aleph$ result to insure correct $\aleph$ propagation. Therefore, **I-Structure Store** semantics are:

Table A.6: Behavior of Operation I-Structure Store	
Input:	**Output:**
$\ll$DATA, $c, i, q, 1, A\gg$	**if** $A = \aleph$
$\ll$DATA, $c, i, q, 2, v\gg$	**then** $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', \aleph\gg$
	else $\Rightarrow \ll$STORE, $A + q_{const}, c, i, q, v\gg$

The action then taken by the structure storage module upon receipt of a store request is:

Table A.7: Behavior of Operation Structure Controller Store	
Input:	**Output:**
$\ll$STORE, $A, c, i, q, v\gg$	$\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', true\gg$

where the "result" *true* is simply an acknowledgement of the completion of the storage request.[†]

Similar definitions define the behavior of the locking operators described in Section 4.3. The semantics of the structure memory controller responses to those requests are also outlined.[‡]

Table A.8: Behavior of Operation I-Structure Lock	
Input:	**Output:**
$\ll$DATA, $c, i, q, 1, S\gg$	**if** $S = \aleph \vee o = \aleph$
$\ll$DATA, $c, i, q, 2, o\gg$	**then** $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p', \aleph\gg$
	else $\ll$LOCK, $S_{base} + o + q_{const}, c, i, q\gg$

Table A.9: Behavior of Operation Structure Controller Lock	
Input:	**Output:**
$\ll$LOCK, $A, c, i, q\gg$	$\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA, $c, i, q', p',$ contents$(A)\gg$

[†]On some tagged-token architectures, notably Monsoon,[61] this acknowledgement is returned when the request is *sent* rather then at the time the request is *acted upon*. In general, this behavior is not acceptable to insure proper termination detection, which Monsoon must then secure by other means.

[‡]These operations are suspiciously similar to the controller fetch and store operations, of course.

Table A.10: Behavior of Operation I-Structure Unlock	
Input:	**Output:**
$\ll$DATA$, c, i, q, 1, A\gg$	**if** $A = \aleph$
$\ll$DATA$, c, i, q, 2, v\gg$	**then** $\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', \aleph\gg$
	else $\ll$UNLOCK$, A + q_{const}, c, i, q, v\gg$

Table A.11: Behavior of Operation Structure Controller Unlock	
Input:	**Output:**
$\ll$UNLOCK$, A, c, i, q, v\gg$	$\forall(q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', true\gg$

More interesting are the semantics of encapsulated (compound) operations. For example, the extant compiler for the ID language treats function application as a single pseudo-instruction **Apply**, as outlined in Figure A.1; a function f of arguments $a_1 \ldots a_n$ is called, with a single result as an answer.[73] This pseudo-instruction must be compiled down into a series of primitive instructions.

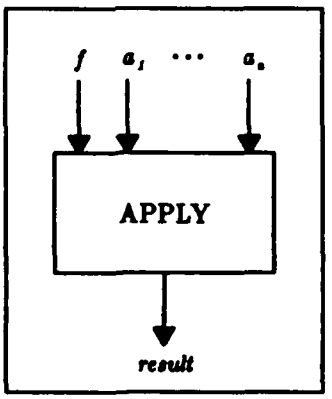


Figure A.1: The Apply Pseudo-Instruction

Previously we have seen only actors which send output tokens to instructions in the same *context* (c, i) as their inputs. To implement the **Apply** pseudo-instruction, we present the first operation which has an output context not equal to the contexts of its inputs. **Change Tag** is described as following:

Table A.12: Behavior of Operation Change Tag	
Input:	**Output:**
$\ll$DATA$, c, i, q, 1, C\gg$	**if** $C \neq \aleph$
$\ll$DATA$, c, i, q, 2, v\gg$	**then** $\Rightarrow \ll$DATA$, c', i', q' + q_{const}, 1, v\gg$
	where $C = (c', i', q')$
	else [None]

In English, the **Change Tag** instruction sends its value input v to a new context, specified by the context input C.

We also need instructions that display conditional behavior in order to support conditional and looping ID expressions. These are supported by a special instruction **Switch**:

Table A.13: Behavior of Operation Switch	
Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v\gg$	**if** $c =$ *true*
$\ll$DATA$, c, i, q, 2, c\gg$	**then** $\forall (q', p') \in q_{true} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else if $c =$ *false*
	then $\forall (q', p') \in q_{false} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else [None]

The reader will note that, unlike other operators, **Switch** instructions have *two* destination lists; one is used for conditions of *false*, and the other for *true*. If the condition is $\aleph$, then no output token is emitted by the **Switch** instruction.

We include a new instruction **No-Value?** (which we will draw as "$= \aleph$") which is used to check for $\aleph$ data (and is therefore not strict in $\aleph$):

Table A.14: Behavior of Operation No-Value?	
Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v\gg$	**if** $v = \aleph$
	then $\forall (q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', true\gg$
	else $\forall (q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i, q', p', false\gg$

In order to make our schemas easier to read, we also introduce a pseudo-instruction $\aleph$?, which is simply a combination of **No-Value?** with a **Switch** as in Figure A.2. As a primitive

instruction, ℵ? would have the following semantics:[†]

Table A.15: Behavior of Operation **Novalue-Switch**

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v\gg$	**if** $c = \aleph$
$\ll$DATA$, c, i, q, 2, c\gg$	**then** $\forall(q', p') \in q_{true} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else $\forall(q', p') \in q_{false} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$

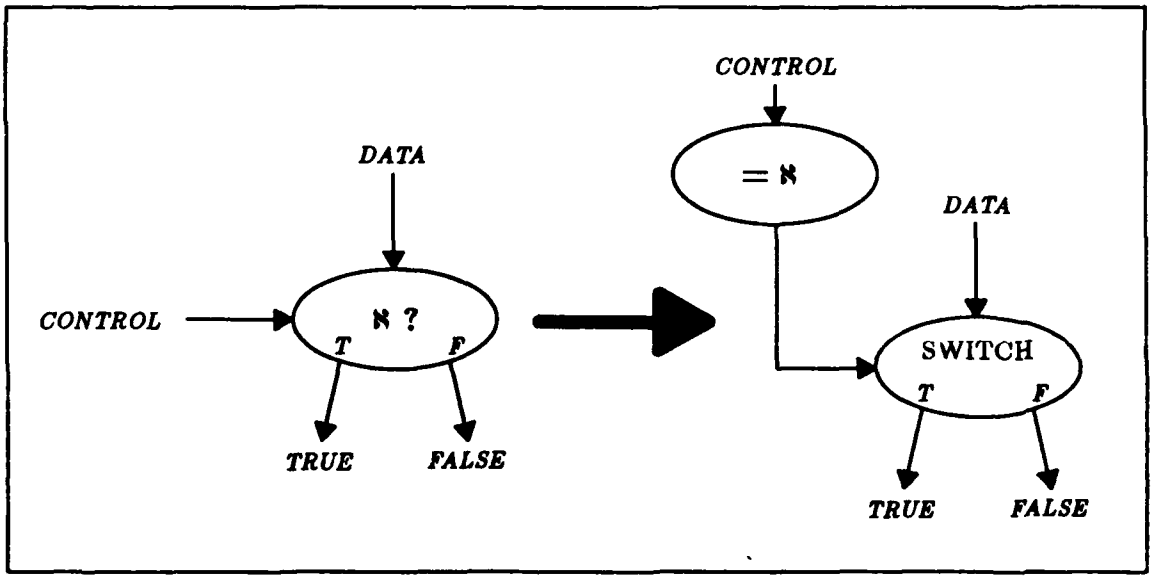


Figure A.2: Schema for Branch on ℵ

Given these instructions, we can now implement the **Apply** pseudo-instruction of Figure A.1 in primitive operations as in Figure A.3.[‡] In this figure, we see that if the output of the **Get Context** instruction is not ℵ, then the result and output signal (to remove the now finished context) are generated by the called procedure itself; otherwise, the procedure is never called and the output result (ℵ) and signal are determined locally. An important detail of this figure is that while function *calling* is strict in ℵ for the function itself (f), *argument* passing is *not* strict in ℵ. Therefore, ID procedures are *not* strict in ℵ.[§]

Schemata also must be found for compiling the other two major encapsulations of the ID language. The first is the ID conditional **if**:

[†]Implementations are of course free to have a primitive ℵ? instruction rather than generating the pair of instructions.

[‡]We have abstracted away, however, the details of context creation and removal, which were explored in Chapter 3.

[§]Nor are arguments to functions strict in any value in ID.

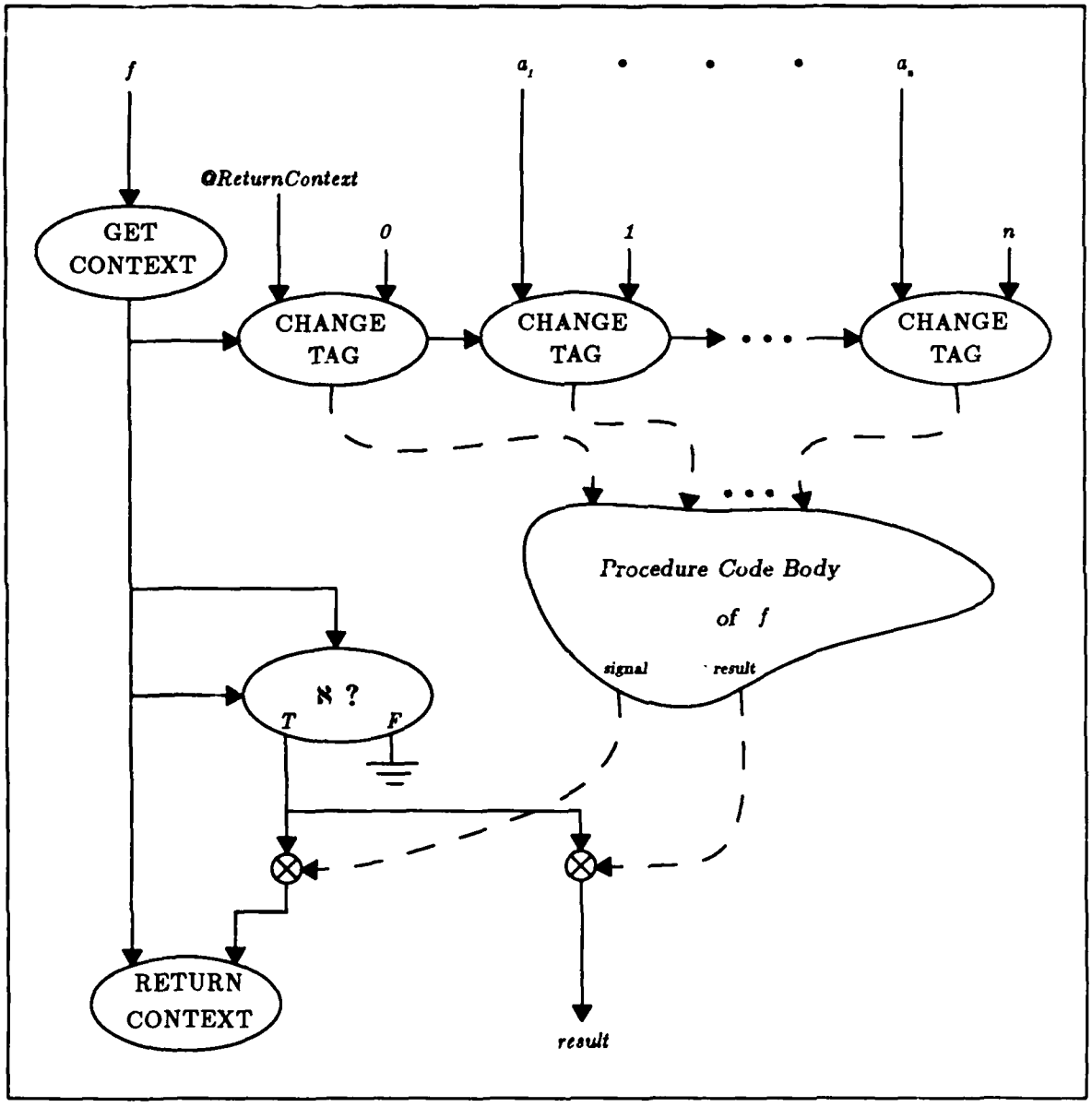


Figure A.3: Schema for Function Calling Accounting for ℵ

if *(Predicate Code Body)*
  then *(TRUE Code Body)*
  else *(FALSE Code Body)*

The schema for **if** blocks, taking into account the possibility of a ℵ conditional, is presented in Figure A.4. In this figure, $input_1 \ldots input_n$ represent the free variables in the two code bodies. The ganged **Switch** instructions send these free variables to the correct code body, and the outputs of the two encapsulated code bodies are then merged. This nondeterministic merge does not cause problems in the determinism of the language, since only *one* of the encapsulated code blocks will send a value to the merge.

The other major encapsulating structure in ID is that created for loops (**while**):[†]

{ **while** *(Predicate Code Body)*
  do *(Loop Code Body)* }

Figure A.5 presents a schema for generating looping execution. In order to support the execution of multiple simultaneous iterations, we need to alter the context information of tokens flowing through loop graphs. This is accomplished with the D and D^{-1} instructions shown in Figure A.5, which manipulate the *iteration* field of the context information.

Both of these operations are unary and trivially strict in ℵ. The former moves a value from one iteration of the loop to the following:

Table A.16: Behavior of Operation D

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v \gg$	$\forall (q', p') \in q_{dest} \Rightarrow \ll$DATA$, c, i+1, q', p', v \gg$

while the latter returns a result from the loop body out into the surrounding context, by clearing the iteration field:

[†]The **for** construct can be simply converted into a **while** loop.

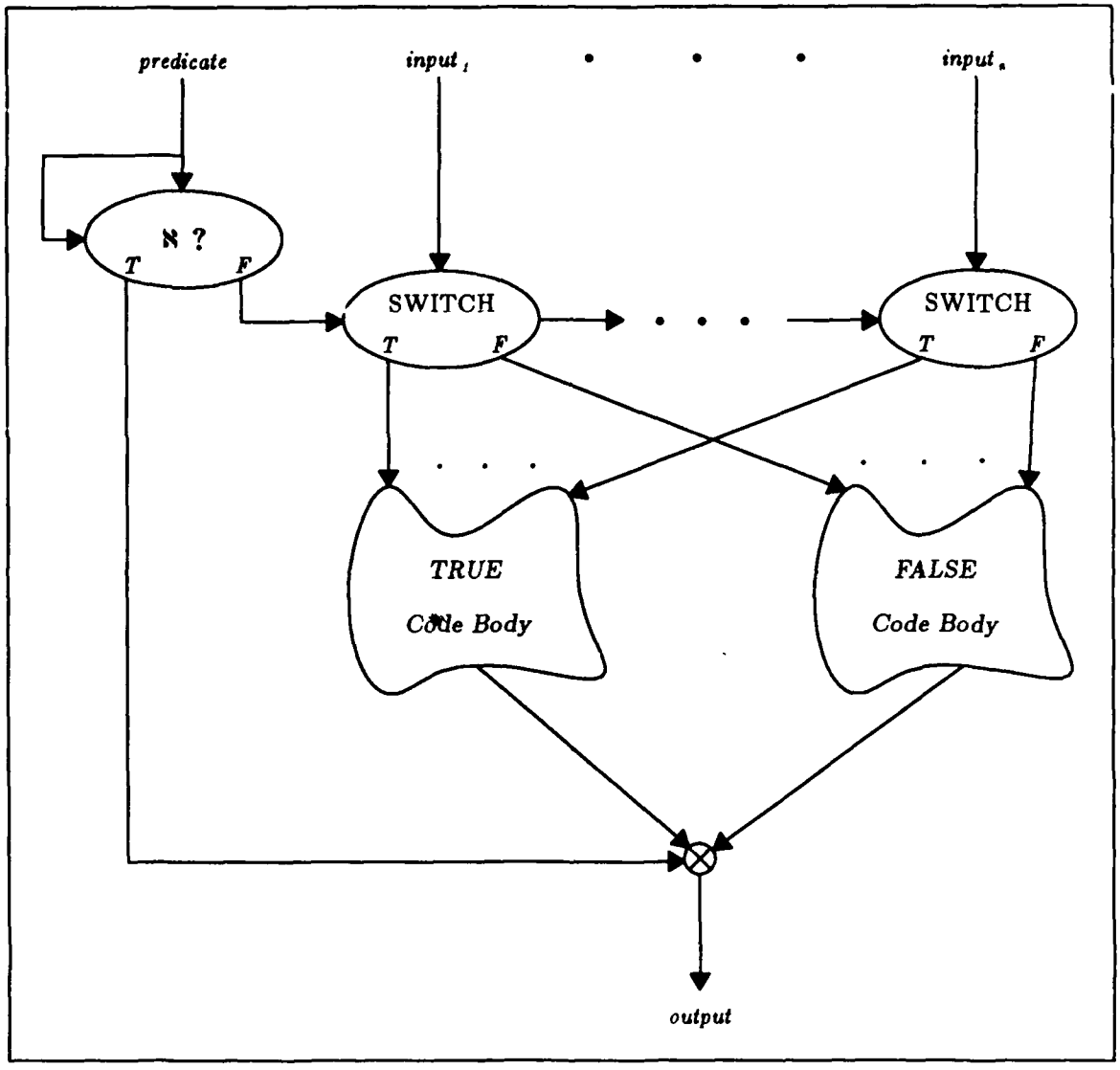


Figure A.4: Schema for Conditionals Accounting for ℵ

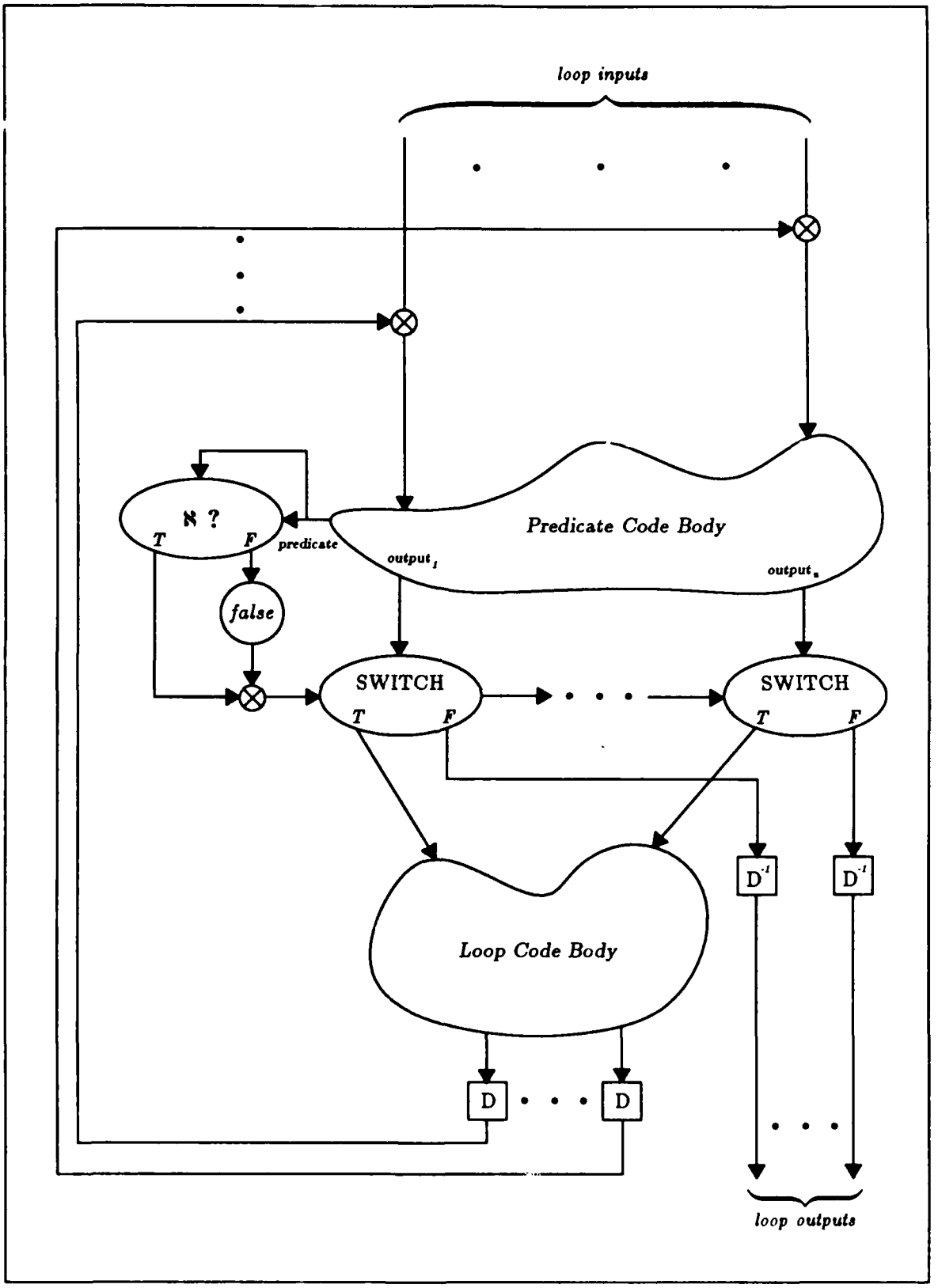


Figure A.5: Schema for Loops Accounting for ℵ

Table A.17: Behavior of Operation $\mathtt{D}^{-1}$	
Input:	**Output:**
$\ll \text{DATA}, c, i, q, 1, v \gg$	$\forall (q', p') \in q_{dest} \Rightarrow \ll \text{DATA}, c, 0, q', p', v \gg$

Now we have a complete set of instructions and schemas for compiling and executing ID code with well-defined propagation of $\aleph$. Unfortunately, however, we have not taken into account the termination of contexts (instantiations of code blocks). In order to do this, the current ID compiler requires every instruction in any encapsulator to contribute to a *signal tree*, the result of which is the *termination signal* of that encapsulator. This information is then used to decide (at run time) when resources of dynamic extent (*e.g.*, related to particular code block instantiations) may be reclaimed. Signal trees are simply built out of trees of the **Gate** instruction we saw in Section 3.2, which have the following dynamic semantics:

Table A.18: Behavior of Operation Gate	
Input:	**Output:**
$\ll \text{DATA}, c, i, q, 1, v \gg$	$\forall (q', p') \in q_{dest} \Rightarrow \ll \text{DATA}, c, i, q', p', v \gg$
$\ll \text{DATA}, c, i, q, 2, c \gg$	

In other words, the gate simply forwards its first (port 1) input to its output destinations. The instruction is strict in $\aleph$ only in its first argument.

With this new information in hand, we can review Figure A.3, Figure A.4 and Figure A.5 (for function calling, conditions and loops respectively) for proper termination. Unfortunately, we find them all deficient exactly in the case of $\aleph$ propagation. Some stronger medicine is necessary.

To that end, we introduce some new semantics for the **Switch** and **Change Tag** instructions, as well as a new three-output instruction **NVSwitch**:

Table A.19: New Behavior of Operation **Change Tag**

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, C\gg$	**if** $C \neq \aleph$
$\ll$DATA$, c, i, q, 2, v\gg$	**then** $\Rightarrow \ll$DATA$, c', i', q' + q_{const}, 1, v\gg$
	where $C = (c', i', q')$
	else $\forall (q', p') \in q_{dest} \Rightarrow \Rightarrow \ll$DATA$, c, i, q', p', \aleph\gg$

The new semantics for **Change Tag** make it in effect a conditional; if the context input C is invalid ($\aleph$), then the **Change Tag** instruction simply sends $\aleph$ to its destination list. On any value context input C, the value v is forwarded into the new context as before. This allows the clearer (and termination-correct) schema for function application found in Figure A.6. The critical path for function application is now no longer than before the introduction of $\aleph$.

Table A.20: New Behavior of Operation **Switch**

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v\gg$	**if** $c =$ *true*
$\ll$DATA$, c, i, q, 2, c\gg$	**then** $\forall (q', p') \in q_{true} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else if $c =$ *false*
	then $\forall (q', p') \in q_{false} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else $\forall (q', p') \in q_{false} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$

These new semantics for **Switch** simply treat $\aleph$ conditionals (c) as if they were false. A quick look at the schema for loop compilation shows that this will be a tremendous simplification, as can be seen in Figure A.7. Again, the critical path for any given loop is now no longer because of the necessary treatment of $\aleph$.

Conditional blocks remain problematic. Although our previous approach to **if** schemas (in Figure A.4) provided correct results, it did not and could not provide correct termination for $\aleph$ conditionals. For proper termination, we need to think of conditionals as three-armed: the *true* arm, the *false* arm and the $\aleph$ arm. To that end we describe the semantics of a new **NVSwitch** instruction:

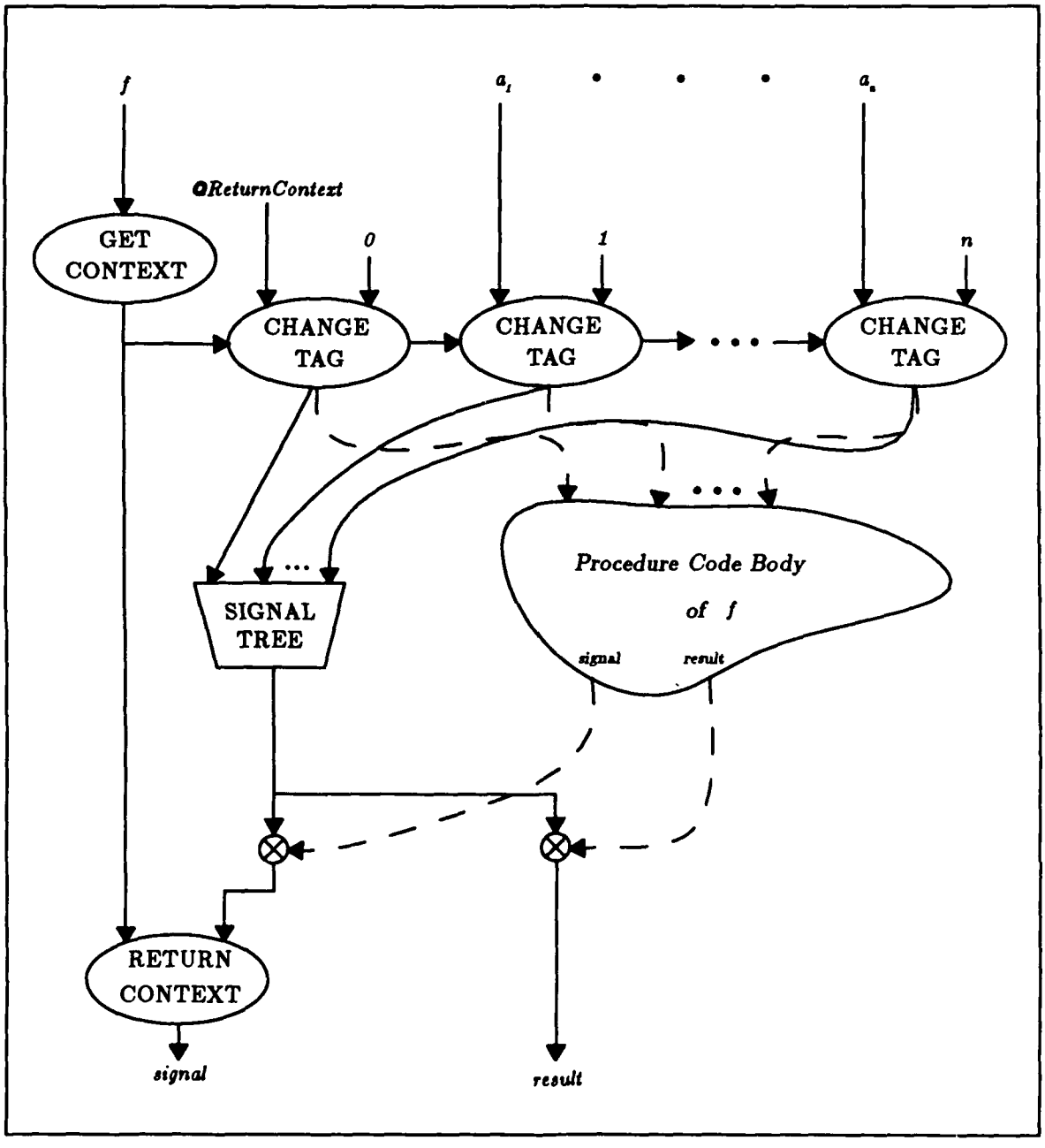


Figure A.6: Schema for Function Application with Proper Termination

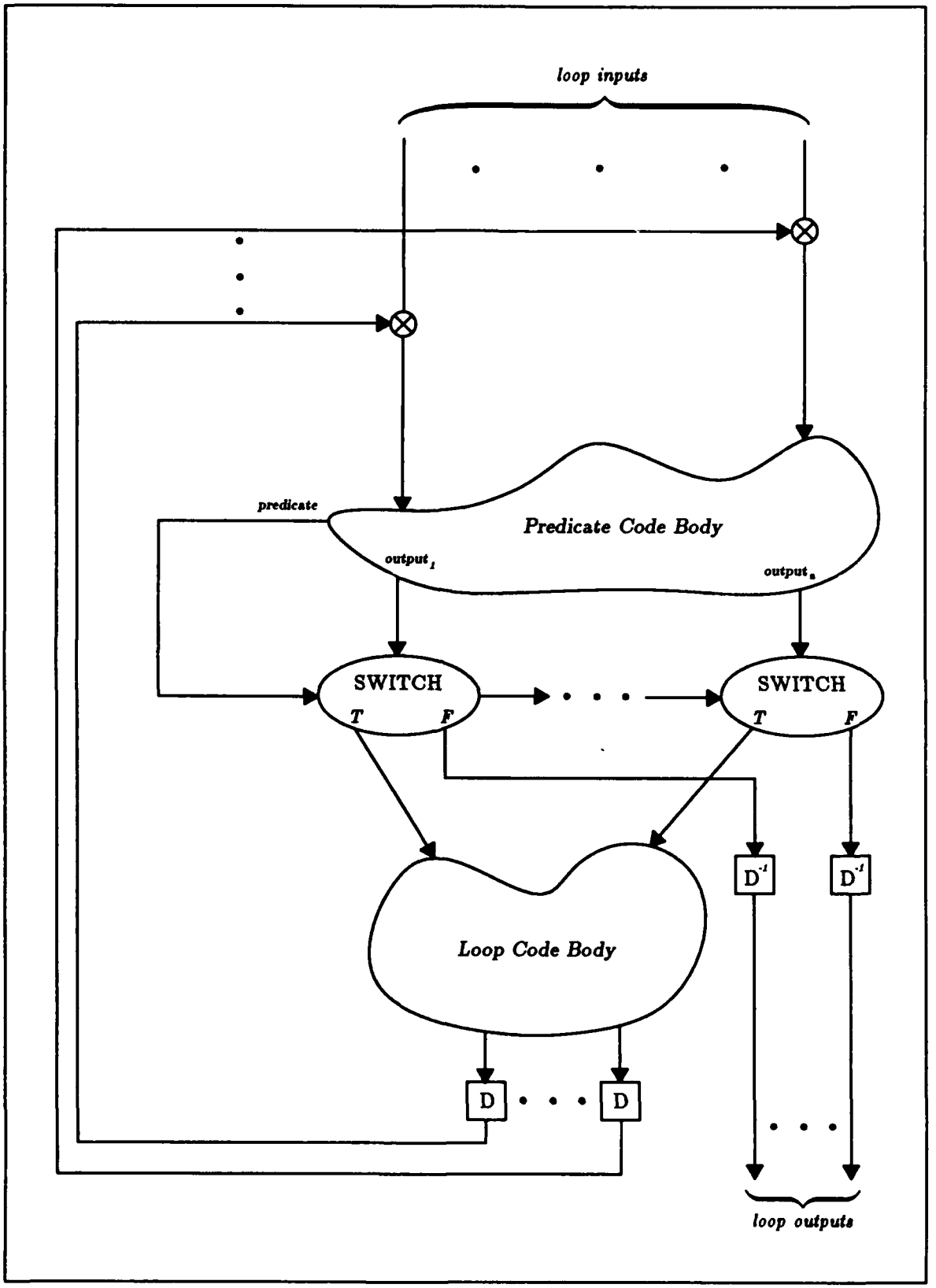


Figure A.7: Schema for Loops with Proper Termination

Table A.21: Behavior of Operation **NVSwitch**

Input:	**Output:**
$\ll$DATA$, c, i, q, 1, v\gg$	**if** $c = true$
$\ll$DATA$, c, i, q, 2, c\gg$	**then** $\forall(q', p') \in q_{true} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else if $c = false$
	then $\forall(q', p') \in q_{false} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$
	else $\forall(q', p') \in q_{\aleph} \Rightarrow \ll$DATA$, c, i, q', p', v\gg$

This new three-destination instruction allows us to directly implement the three-armed conditional as in Figure A.8. It should also be noted that we could simply ignore treatment of $\aleph$ in conditionals and generate a two-armed conditional with standard **Switch** instructions; the run-time overhead of this approach would simply be the overhead of propagating $\aleph$ through the *false* code block of the conditional. This trade-off can be made by the machine architect (in deciding whether to implement three-destination instructions) and/or the ID compiler writer (in deciding whether to generate a two- or three-armed conditional block, or which arm of the conditional to take on $\aleph$ input).

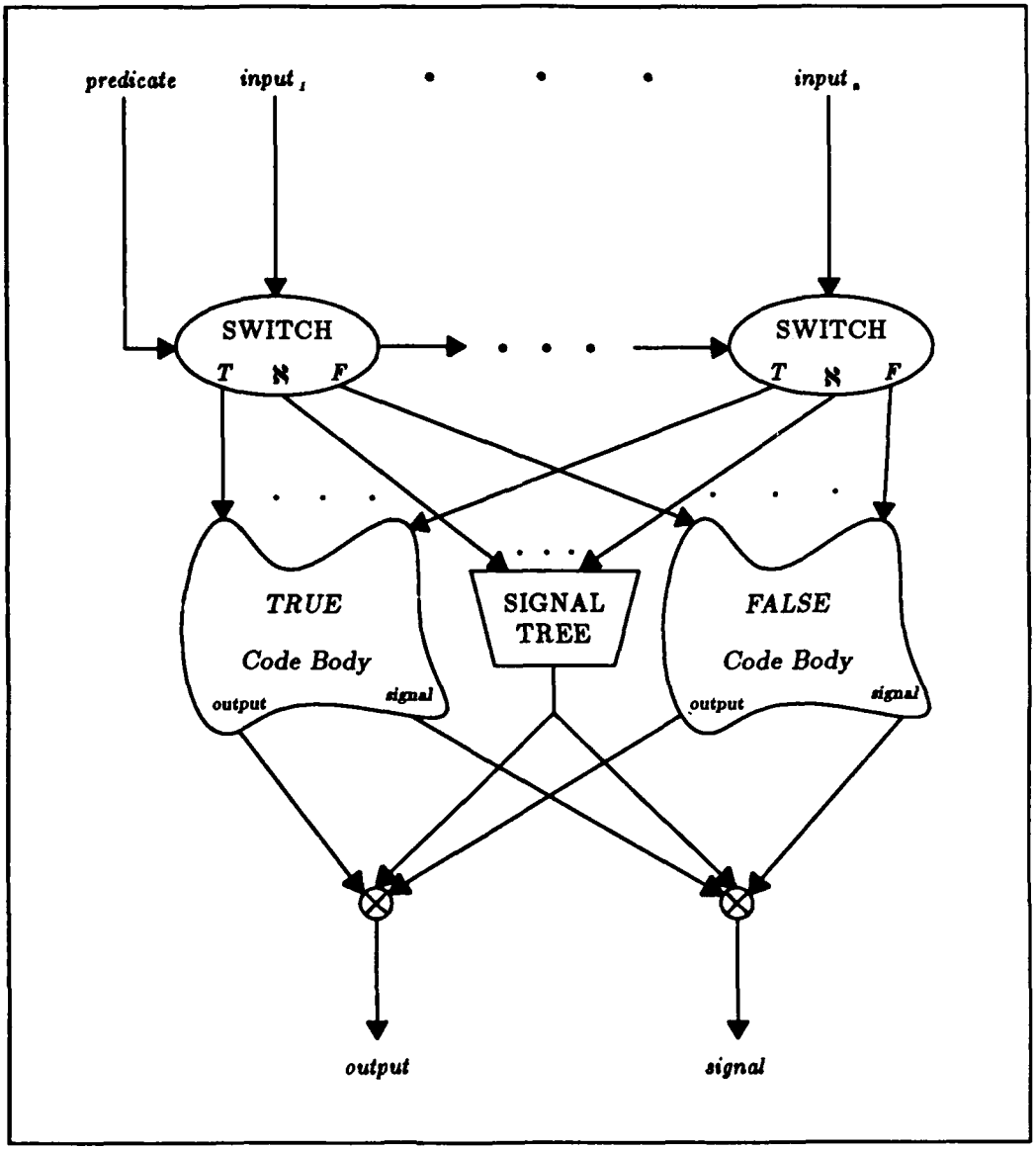


Figure A.8: Schema for Conditionals with Proper Termination

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge Massachusetts, 1984.

[2] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press Series in Artificial Intelligence. MIT Press, Cambridge Massachusetts, 1986.

[3] Khayri A. M. Ali. OR-Parallel Execution of Prolog on a Multi-Sequential Machine. *International Journal of Parallel Programming*, 15(3):189–214, June 1986.

[4] Mark Anderson and Francine Berman. Removing Useless Tokens from a Dataflow Computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 614–617, August 1987.

[5] Arvind and J. Dean Brock. Resource Managers in Functional Programming. *Journal of Parallel and Distributed Computing*, 1(1), January 1984.

[6] Arvind and David E. Culler. Dataflow Architectures. In *Annual Review of Computer Science, Volume 1*, pages 225–253. Annual Reviews, Inc., Palo Alto, California, 1986.

[7] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Fifth Generation Computer Architectures*, pages 103–121. Elsevier Science Publishers B.V., 1986.

[8] Arvind and Kattamuri Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5(5), October 1988.

[9] Arvind, K. P. Gostelow, and W. Plouffe. An Asynchronous Programming Language and Computing Machine. Technical Report 114, University of California, Irvine, Department of Information and Computer Science, Irvine California, December 1978.

[10] Arvind, Kim P. Gostelow, and Wil Plouffe. Indeterminacy, Monitors, and Dataflow. In *Proceedings of Sixth ACM Symposium on Operating Systems Principles*, pages 159–169, November 1977.

[11] Arvind and R. A. Iannucci. Instruction Set Definition for a Tagged-Token Data Flow Machine. Computation Structures Group Memo 212-3, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, February 1983.

[12] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. Id Nouveau: Reference Manual. Computation Structures Group Memo 284, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, 1987.

[13] Arvind and Rishiyur Sivaswami Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference*, number 259 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, June 1987. Springer-Verlag. (Also to appear in IEEE Transactions on Computers).

[14] Arvind, Rishiyur Sivaswami Nikhil, and Keshav Kumar Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction*, number 279 in Lecture Notes in Computer Science, pages 336–369, Santa Fe, New Mexico, September/October 1987. Springer-Verlag. (Also to appear in TOPLAS).

[15] Henry G. Baker Jr. and Carl Hewitt. The Incremental Garbage Collection of Processes. A. I. Memo 454, M.I.T Artificial Intelligence Laboratory, Cambridge Massachusetts, December 1977.

[16] Paul S. Barth. Managing Nondeterministic Access to Shared Resources in a Dataflow System. Class paper for M.I.T. class 6.891, May 1988.

[17] Paul S. Barth. Managers, state-sensitive computation, and the lock protocol. Personal communication, April 1989.

[18] Paul S. Barth and Rishiyur S. Nikhil. Supporting State-Sensitive Computation in a Dataflow System. Computation Structures Group Memo 294, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, March 1989.

[19] Nena B. Bauman and Robert A. Iannucci. A Methodology for Debugging Data Flow Programs. Computation Structures Group Memo 219, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, October 1982.

[20] Gerard Berry and Jean-Jacques Levy. Minimal and Optimal Computations of Recursive Programs. In *Proceedings of the 1977 Conference on Principles of Programming Languges*, January 1977.

[21] Ronald J. Brachman and Hector J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *Association for the Advancement of Artificial Intelligence*, pages 34–37, August 1984.

[22] J. Dean Brock and Arvind. Streams and Managers. Computation Structures Group Memo 217, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, 1982.

[23] A. J. Catto and J. R. Gurd. Resource Management in Dataflow. In *Proceedings of the 1981 Comference on Functional Programming Languages and Computer Architecture*, pages 77–84. ACM, October 1981.

[24] Philip H. Chu. Exploiting Parallelism in Game-Playing Programs. Bachelor's thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1988.

[25] Keith Clark and Steve Gregory. PARLOG: Parallel Programming in Logic. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, chapter 6, pages 109–130. Kluwer Academic Publishers, Norwell Massachusetts, 1988.

[26] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 2nd edition, 1984.

[27] Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California. *Unix Programmer's Manual*, August 1983.

[28] David E. Culler. *Effective Dataflow Execution of Scientific Applications.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1989.

[29] Haskell B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[30] Doug DeGroot. Restricted AND-Parallel Execution of Logic Programs. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, chapter 5, pages 91–107. Kluwer Academic Publishers, Norwell Massachusetts, 1988.

[31] E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, London, 1968.

[32] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[33] L. D. Erman. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainties. *Computing Surveys*, 12(2):213–253, 1980.

[34] Scott E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press, Cambridge Massachusetts, 1979.

[35] Richard P. Gabriel and John McCarthy. Queue-based Multiprocessor Lisp. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[36] Richard P. Gabriel and John McCarthy. QLisp. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, chapter 4, pages 63–89. Kluwer Academic Publishers, Norwell Massachusetts, 1988.

[37] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Sprinter-Verlag, New York, 1979.

[38] Robert H. Halstead, Jr. MultiLisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages*, 7(4):501–538, October 1985.

[39] Robert H. Halstead, Jr. Parallel Computing using MultiLisp. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, chapter 2, pages 21–49. Kluwer Academic Publishers, Norwell Massachusetts, 1988.

[40] Robert H. Halstead, Jr. and Juan R. Loaiza. Exception Handling in MultiLisp. In *Proceedings of the 1985 International Conference on Parallel Processing*, University Park, Penn., August 1985.

[41] Steven K. Heller. An I-Structure Memory Controller. Master's thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1983.

[42] Steven K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, February 1989.

[43] P. Henderson and J. H. Morris. A Lazy Evaluator. In *Proceedings of the 3rd ACM Symposium on the Principles of Programming Languages*, pages 95–103, 1976.

[44] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge Massachusetts, 1985.

[45] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, August 1978.

[46] Bharat Jayaraman and Robert M. Keller. Primitives for Resource Management in a Demand-Driven Reduction Model. *International Journal of Parallel Programming*, 15(3):215–244, 1987.

[47] Donald E. Knuth. An Analysis of Alpha-Beta Pruning. Technical Report CS-74-441, Stanford University Computer Science Department, Stanford California, August 1974.

[48] Donald E. Knuth. *The TEXbook.* Addison-Wesley Publishing Company, Reading, Massachusetts, October 1987.

[49] William Arthur Kornfeld. *Concepts in Parallel Problem Solving.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, 1981.

[50] William Arthur Kornfeld. The Use of Parallelism to Implement a Heuristic Search. A. I. Memo 627, M.I.T Artificial Intelligence Laboratory, Cambridge Massachusetts, March 1981.

[51] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocessor Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[52] Leslie Lamport. LATEX: *A Document Preparation System.* Addison-Wesley Publishing Company, Reading, Massachusetts, April 1986.

[53] Barbara Liskov, Eliot Moss, C. Shaffert, and B. Scheiffer. CLU Reference Manual. Computation Structures Group Memo 161, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, July 1978.

[54] Patrick F. McGehearty and Edward J. Krall. Execution of Common Lisp Programs in a Parallel Environment. In Janusz S. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, chapter 3, pages 51–62. Kluwer Academic Publishers, Norwell Massachusetts, 1988.

[55] Dinarte R. Morais. Id World: An Environment for the Development of Dataflow Programs Written in Id. Technical Report 365, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, May 1986.

[56] Rishiyur S. Nikhil. Practical Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures*, pages 319–333, Nancy, France, September 1985. Springer-Verlag.

[57] Rishiyur S. Nikhil. Id World Reference Manual. Computation Structures Group Memo, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, April 1987.

[58] Rishiyur S. Nikhil and Arvind. Id Nouveau. Computation Structures Group Memo 265, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, 1986.

[59] Nils J. Nilsson. *Principles of Artificial Intelligence.* Morgan Kaufmann, 1980.

[60] Randy Osborne. *Speculative Computation in Multilisp: A Model and its Implementation.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, August 1989. (Expected).

[61] Gregory M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, August 1988.

[62] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing, Reading, Massachusetts, 1984.

[63] Keshav Pingali and Arvind. Efficient Demand-Driven Evaluation, Part I. *ACM TOPLAS*, 7(2):311–333, 1985.

[64] Wilfred Edmund Plouffe, Jr. *Exception Handling and Recovery in Applicative Systems.* Ph.D. thesis, University of California at Irvine, Irvine, California, 1980.

[65] M. Raynal. *Algorithms for Mutual Exclusion.* Scientific Computation Series. MIT Press, Cambridge Massachusetts, 1986. Originally published under the title *Algorithmique du parallélisme* by Dunod informatique, France.

[66] Richard Mark Soley. Implicit Serialization in Dataflow Programs: How to Support Input/Output in Id. Computation Structures Group Memo 277, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, 1987.

[67] Richard Stallman, Daniel Weinreb, and David Moon. *Lisp Machine Manual.* M.I.T Artificial Intelligence Laboratory, 6^{th} edition, June 1984.

[68] Kenneth M. Steele. Implementation of an I-Structure Memory Controller. Master's thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, December 1989. (Expected).

[69] Kenneth M. Steele and Richard Mark Soley. Virtual Memory on a Dataflow Computer. Computation Structures Group Memo 289, M.I.T. Laboratory for Computer Science, Cambridge Massachusetts, July 1988.

[70] Guy Lewis Steele Jr. *Common Lisp the Language.* Digital Press, 1984.

[71] Guy Lewis Steele Jr. and Gerald J. Sussman. The Art of the Interpreter or, The Modularity Complex. A. I. Memo 453, M.I.T Artificial Intelligence Laboratory, Cambridge Massachusetts, May 1978.

[72] Isabel T. Szabó. *Illustrate: User Manual.* A. I. Architects, Inc., Cambridge, Massachusetts, August 1986.

[73] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Master's thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1986.

[74] Kenneth R. Traub. *Sequential Implementation of Lenient Programming Languages.* Ph.D. thesis, Massachusetts Institute of Technology, Cambridge Massachusetts, May 1988.

[75] K. Ueda. *Guarded Horn Clauses.* Ph.D. thesis, University of Tokyo, Tokyo Japan, 1985.

[76] Stephen A. Ward. Functional Domains of Applicative Languages. Technical Report 136, M.I.T. Project MAC, Cambridge Massachusetts, September 1974.

[77] David E. Wilkins. Using Knowledge to Control Tree Searching. *Artificial Intelligence*, 18(1):1–51, January 1982.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy